DOCUMENT RESUME

ED 057 828 LI 003 325

AUTHOR	Parker, Edwin B.
TITLE	SPIRES (Stanford Public Information Retrieval System) 1970-71 Annual Report.
INSTITUTION	Stanford Univ., Calif. Inst. for Communication Research.
SPONS AGENCY	National Science Foundation, Washington, D.C. Office of Science Information Services.
PUB DATE	Dec 71
NOTE	154p.; (23 References)
EDRS PRICE	MF-$0.65 HC-$6.58
DESCRIPTORS	*Computer Programs; *Information Retrieval; *Information Storage; *Information Systems; *On Line Systems
IDENTIFIERS	Computer Software; SPIRES; *Stanford Public Information Retrieval System

ABSTRACT

SPIRES (Stanford Public Information REtrieval System) is a computer information storage and retrieval system being developed at Stanford University with funding from the National Science Foundation. SPIRES has two major goals: to provide a user-oriented, interactive, on-line retrieval system for a variety of researchers at Stanford; and to support the automation efforts of the university libraries by developing and implementing common software. SPIRES I, a prototype system, was implemented at the Stanford Linear Accelerator Center (SLAC) in 1969, from a design based on a 1967 information study involving physicists at SLAC. Its primary data base is a high-energy-physics preprints file. Evaluation of SPIRES I resulted in the definition of a production information storage and retrieval system, SPIRES II. This system will be available daily, beginning in mid-1972, to faculty, staff, and students of the University. It is characterized by flexibility, simplicity, and economy. SPIRES II will operate on-line on an IBM 360/67 computer. This report summarizes the uses of the SPIRES I system over the past year and describes both the nature of SPIRES II and this system's development over the past year. (Author)

ED057828

U.S. DEPARTMENT OF HEALTH,
EDUCATION & WELFARE
OFFICE OF EDUCATION
THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL OFFICE OF EDUCATION POSITION OR POLICY

(2) SPIRES

(Stanford Public Information REtrieval System)

1970-71 Annual Report.

to

(4) the National Science Foundation, Wash
(Office of Science Information Services) BBB01121

(1) Edwin B. Parker
Principal Investigator

(3) Insti~~tute~~ for Communication Research GI98-2550
Stanford University, Calif

(5) December 1971

LI 003 325

154p.

ABSTRACT

SPIRES (Stanford Public Information REtrieval System) is a computer information storage and retrieval system being developed at Stanford University with funding from the National Science Foundation. SPIRES has two major goals: to provide a user-oriented, interactive, on-line retrieval system for a variety of researchers at Stanford; and to support the automation efforts of the university libraries by developing and implementing common software.

SPIRES I, a prototype system, was implemented at the Stanford Linear Accelerator Center (SLAC) in 1969, from a design based on a 1967 information study involving physicists at SLAC. Its primary data base is a high-energy-physics preprints file. Evaluation of SPIRES I resulted in the definition of a production information storage and retrieval system, SPIRES II. This system will be available daily, beginning in mid-1972, to faculty, staff, and students of the University. It is characterized by flexibility, simplicity, and economy. SPIRES II will operate on-line on an IBM 360/67 computer. This report summarizes the uses of the SPIRES I system over the past year and describes both the nature of SPIRES II and this system's development over the past year.

CONTENTS

1.0 BACKGROUND 1

2.0 SPIRES I IN THE PAST YEAR 3

3.0 SPIRES II DEVELOPMENT 5

3.1 The Computing Environment. 5
3.2 Development Status. 7
3.3 Capabilities and Services Planned. 10
3.4 Work in the Coming Year. 12

4.0 SPIRES AND BALLOTS 14

APPENDICES

A. List of Selected Publications and Reports Relating to SPIRES

B. Contents Pages for Requirements for SPIRES II

C. Design of SPIRES II, Volume 1

D. Preprints/Anti-preprints

1.0 Background

This is the fourth annual report to the National Science Foundation on Project SPIRES (Stanford Physics Information REtrieval System), now known as the Stanford Public Information REtrieval System. The first, 1967 Annual Report <11>* described the results of a behavioral information study of a target population of physicists; the 1968 Annual Report <12> documented the SPIRES I on-line programming; the 1969-70 Annual Report <13> described the operation and evaluation of SPIRES I and the development plan for SPIRES II, the production system. This report describes the current operation of SPIRES I and the progress made in developing SPIRES II during the period July 1, 1970, to June 30, 1971.

SPIRES has two long-range goals. The first is to provide a user-oriented, interactive, production on-line information storage and retrieval system for a variety of research groups in the Stanford community. The second is to support the automation efforts of university libraries (Project BALLOTS) <8, 15> by contributing to common software development. An immediate short-range goal has been to provide an on-line bibliographic information service for Stanford physicists, particularly for high-energy physicists. All of these goals must be achieved within a framework of effective, efficient operations. Effectiveness is ensured by carefully studying and constantly interacting with users and the user environment. Efficiency is ensured by evaluating costs and performance factors under operating conditions.

In 1967, a comprehensive user study was conducted on a target population of physicists. This study established information needs and priorities as a basis for system design (see the 1967 SPIRES Annual Report). In late 1967, a small, one-terminal demonstration system was installed on the 360 model 75 computer (since replaced by a 360/91) at the Stanford Linear Accelerator Center (SLAC), using an IBM 2250 display terminal. Following the demonstration of the pilot system, most of 1968 was spent in creating the software necessary for a multiple-user on-line system. This included the development of an on-line supervisor program (see the 1968 SPIRES Annual Report), and of search, retrieval, and update programs. By early 1969, SPIRES I had been tested and was ready for service; in late February operation began for an hour and a half a day, five days a week. This service schedule continued through the summer of 1969. IBM 2741 typewriter terminals were placed in the Stanford University

---

*Number in brackets refer to Appendix A, a list of publications and reports relating to SPIRES.

Libraries and in the SLAC Library. (The SPIRES system, however, can be used from any terminal on campus.)

After several months of operational experience, the last quarter of 1969 was spent in evaluating the SPIRES I system (see the 1969-70 SPIRES Annual Report). This evaluation was conducted by members of the SPIRES staff with the assistance of an independent computer consultant, Robert L. Patrick. It indicated that with the successful operation of SPIRES I a major milestone had been reached. Technical feasibility was clearly demonstrated. The special target audience of high-energy physicists had found the SPIRES system useful. Another user group (the Library staff), with almost no knowledge of computers, had been able to use the system after only a short training period. Various data bases had been created and successfully searched concurrently from different points on campus. The evaluation revealed that the data bases used by the SPIRES system, particularly the library files and special subject files such as a physics preprint file, are characterized by continued growth and intensive updating. If the SPIRES system were in use full time, its users would have to be assured of software and hardware reliability.

The experience with SPIRES I was the basis for a six-phase development cycle defined for a SPIRES II production system. The first phase of the SPIRES II system development process was completed during the first quarter of 1970. This phase of preliminary analysis produced a major document <15> that characterized the users and the user environment and summarized the limitations of SPIRES I; it then went on to outline a long-range scope of retrieval and file management capabilities as well as the first implementation of SPIRES II.

The second phase, detailed analysis, was a period of crucial activity. System requirements (such as performance and output documentation) were established and approved by project staff and system users. A variety of technical tasks were carried out: the evaluation of existing programming languages and software, system simulation, the writing of an on-line command language, the designing of an analyzer to parse the language. This activity was well underway in July 1970, the beginning of this reporting year.

2.0 SPIRES I IN THE PAST YEAR

For most of the reporting year, 1970-71, SPIRES I continued in operation on the Campus Facility 360/67, where it was used primarily by SLAC physicists and Library staff. During this period, however, work was begun to move SPIRES I to the IBM 360 model 91 computer at SLAC. By June 1971, the following components of SPIRES I were successfully operating on the 360/91:

- On-Line Retrieval
- Batch Build (Data Base and Indexes)
- SLAC Publication List production (by author, subject, and number)
- Weekly Preprint List (PPF) production
- Anti-Preprint List Production
- Data Base Checkpoint/Restore
- Data Base Recovery/Reconstruct
- Miscellaneous Diagnostic Routines

SPIRES I is presently accessible on the 360/91 from any SLAC terminal and from terminals on campus and elsewhere that have telephone dialup and entree to a SLAC account. Operation of the system requires mounting one disk pack. A 15-minute search utilized about .0004/min CPU time. It uses just under 300,000 bytes of core storage. Searches typically bring the system up for five- to twenty-minute periods and are coordinated through the SLAC Library to minimize the use of system resources.

The SLAC Library's relation to SPIRES has continued to be that of an experimental user group. Since 1968, a SLAC Librarian, L. Addis, has acted as liaison to SPIRES and as coordinator of the Library's SPIRES-related activities including on-line development of the principal SPIRES I database, a file of some 14,000 high energy physics preprints. During 1970-71, the library has continued weekly maintenance and updating of the preprint database (now on the 360/91), as well as weekly publication of the bulletin "Preprints in Particles and Fields (PPF)". PPF, with its companion ANTIPREPRINTS, which has been published since 1969 (see SPIRES Annual Report 1969-70 and Appendix D), became self-supporting in July 1970 when the AEC seed-money was exhausted. By the end of FY71, PPF had more than 600 subscribers (not including those at SLAC) at the rates of \$10.00/year in the U.S., Canada, and Mexico, and \$18.50 for overseas airmail.

By June 1971, arrangements were underway to supply weekly tapes of preprint information, in SPIRES input format, to the California Institute of Technology, the University of Texas, and to the Deutsches Elektronen-Synchrotron in Hamburg. (R. Parsons at the University of Texas has plans, for instance, for studies utilizing the citation index feature). If experimental tape distribution proves successful, it may be feasible to offer tapes to the other high-energy physics facilities which have expressed interest. Tapes are currently written either 7 track or 9 track, 800 BPI, and with or without citations. Users supply minitapes.

At the suggestion of the American Physical Society Division of Particles and Fields, arrangements are now underway to include data on current experimental high-energy physics proposals, in the SPIRES database and periodically in PPF. The Lawrence Berkeley Laboratory Particle Data Group utilizing SLAC Library files (set up, by the way, in response to suggestions in the 1968 SPIRES interviews) is analyzing available experimental proposal information for beam composition, detection method, number of events, etc. The Particle Data Group also plans to use the resulting SPIRES input tapes in their own compilations.

The building of a complete database utilizing the DESY High Energy Physics Index tapes is underway. The size of previous DESY files has been limited by the fact that two additional disk packs are required for the operation, one for the 1968-69 files and another for the more current files. When the DESY file is complete it will contain approximately 40,000 to 50,000 items dated from 1968 to the present. It will be updated fortnightly and searchable by author, title, date, and, most importantly, keyword. In the meantime, other facilities are experimenting with possible use of the DESY tapes in the SPIRES format. Dr. K. Mellentin at DESY has given permission for such experimentation, the SPIRES format being a particularly convenient one for users.

Together, the preprint and DESY databases are expected to provide on-line access through SPIRES I to virtually all high energy physics literature, published and unpublished, including current experimental proposals, from 1968 to the present.

During the 1971-72 reporting period, SPIRES I activities are expected to emphasize the completion of the DESY database, consolidation and correction of the SPIRES input archive tapes for possible use by other facilities, corrections to the current preprint database, as well as continuation of current production schedules. Alternative plans for the utilization of SPIRES II by SLAC users will be developed.

3.0 SPIRES II DEVELOPMENT

3.1 The Computing Environment

SPIRES I was originally implemented on the Campus Facility IBM 360 model 67. During 1969 and early 1970, the Campus Facility machine was close to saturation. The installation software at that time was workable and efficient, but had not yet been fully optimized. Furthermore, there was a heavy batch workload. Therefore, a decision was made to turn to a machine (of the model 50 class) outside the Campus Facility in support of campus information retrieval. But the restrictive economic environment later caused that decision to be rescinded. Meanwhile, two things occurred in the Campus Facility: a gain in CPU cycle availability due to substantial software optimization, and a decrease in the overall workload. In the past year, reliability on the 360/67 has increased to the point where uptime is around 96 percent. Throughput in the high-speed batch partition has improved 40 percent. For example, the execution time for an average job has been reduced from 4.3 seconds to 2.2 seconds and the minimum job cost has been reduced from fifty cents to twenty-five cents. Text-editor (WYLBUR) throughput has increased 100 percent--i.e., it has doubled, effectively cutting costs to the user by 50 percent. These improvements to the operation of the 360/67 resulted in an average machine cycle availability of 30 percent.

These facts, coupled with the University's desire to make maximum use of its available computer resources, dictated a SPIRES II implementation on the Campus Facility computer. The technical staff of the Campus Facility agreed to make the necessary modifications to the installation software to support a reliable on-line system, and to aid the development and installation of the system, the SPIRES group became an integral part of the Campus Facility systems group.

In order to accommodate SPIRES, the following hardware is being added to the Campus Facility configuration.

1. Disk Storage Drives. When the SPIRES II system becomes available to the campus community, there will be additional storage devices provided. These devices will provide adequate and fast-access storage for data bases. Each drive will provide storage space for approximately 28-million characters.

2. A PDP-11 Front-end Computer. All communications between SPIRES and the CRT terminals will be via a PDP-11 "front-end" machine. This will, among other things, provide faster communications for the terminals.

3. Terminal Equipment. SPIRES will be available via an upper/lower-case CRT terminal (Sanders Associates 800 series) and a less expensive upper-case only CRT Terminal (The Hazeltine 2000). These terminals provide fast, silent display of large amounts of data. It is estimated that the proposed configuration will be able to support up to 32 CRT terminals concurrently; if that number is exceeded, a second PDP-11 can be added to the configuration.

In the Campus Facility 360/67 system software there are six major partitions: the operating system, high-speed batch/FUTIL, large batch, the Stanford time-sharing monitor (ORVYL), the WYLBUR text editor, the MILTEN terminal communicator, and the HASP spooling processor. ORVYL operates in a 220,000-byte partition. It enables a program to reside in segments on a drum, and to share core memory with other executing programs. Such programs are called subprocessors. They are re-entrant; thus many terminal users can use a subprocessor concurrently. ORVYL is the time-sharing monitor performing such functions as reading necessary program segments into core from the drum, and writing user work areas out to the drum when another user is being serviced. The monitor further decides which user should be serviced next.

SPIRES II will operate as a subprocessor under ORVYL, as LISP and BASIC do currently. A simple simulation has predicted good response time, with little detrimental effect on the rest of the system.

MILTEN, the communications monitor, currently supports 88 2741 typewriter terminals. Modifications to meet the needs of SPIRES include supporting CRT display terminals using the PDP-11 computer as a line-handling and intermediate buffering device. MILTEN allows the user to connect to a time-sharing subprocessor or to WYLBUR, the on-line Text Editor. One of the functions of the terminal system is that the user can use WYLBUR text-editing commands transparently while connected to an ORVYL subprocessor and vice versa just by typing the command verb required.

As indicated above, the project decided that to implement SPIRES II as an ORVYL subprocessor would require modifications to the Campus Facility software system. The Stanford Computation Center has made the necessary expertise available at no cost to the project to accomplish the following modifications <9>.

1. ORVYL File System. The present file system under ORVYL provides excellent support for small user files, which are transient in nature and are easily re-created if a system failure occurs. But present Campus Facility requirements double the

input-output load in handling large files. An extension is therefore being provided to the ORVYL file system that will permit rapid access to large files and that will ensure file protection.

2. Subprocessor Communications Area. An ORVYL subprocessor is re-entrant, and is written as if only one user existed. Thus the subprocessor, while acting on behalf of one user, may not have access to the workspace of any other user. It is therefore not possible at present for the subprocessor to "remember" anything as it goes from user to user. To remedy this, a resident area is being provided for each subprocessor in the system. Here the subprocessor can store the work it is doing for one user whenever it goes to another user. It will be impossible for any material in this area to be simultaneously modified (updated) by more than one user.

3. Virtual Access Method (VAM). Provision is being made for programs running in a batch partition to access any ORVYL disk data sets, allowing on-line file access modules to be used in batch mode with no source code modification. Furthermore, the batch programs may be run whether ORVYL is executing at the time or not. The only restriction (which turns out to be a benefit with respect to security and reliability) is that no more than one such batch program may be executed in the system at one time.

4. CRT Terminal Minicomputer Support. ORVYL and MILTEN will be modified to allow CRT terminal support via the front-end PDP-11.

3.2 Development Status

The SPIRES 1969-70 Annual Report describes a six-phase development process being followed. At that writing, the project was involved in the second phase, detailed analysis. In the reporting period 1970-71, detailed analysis was completed; phase three, general system design, was accomplished; and phase four, detailed system design and programming, was begun. (The last two phases, implementation and installation, will be accomplished during 1971-72.) The phases have shifted and overlapped in actual practice, so that general and detailed design intermingled and detailed design has involved some of the coding and testing described as part of phase five, implementation.

In the third quarter of 1970, a general file structure design for SPIRES II was prepared. This was refined and documented in the following quarters. The design was for a multiple-indexing structure that allows both full and partial data records to be stored and accessed. Access (index) records and goal (data) records can be stored together or separately, whichever method is most efficient. Redundancy is built into the file to ensure reliability. Access record values are generated from goal records, so that the access records can be re-created if necessary. Input and output access time is minimized by providing for frequently used tables and dictionaries to be held in core storage. Information common to the records in a file is stored in one spot for economy of space. Transaction logging collects information for recovery procedures, if necessary, and for resource accounting.

Reliability is a major goal of SPIRES II, and general specifications for the recovery techniques needed to ensure reliability were also written at the beginning of the reporting year. Recovery is accomplished by reapplying copies of the deferred update queue, accumulated on tape since the last full dump.

Early in the reporting period, the external specifications for the SPIRES II on-line command language were completed. These specifications described the language from the user's point of view--for example, they contained descriptions of how individual commands were to be used to perform required functions. The on-line commands covered four areas: file definition, file update and maintenance, search and retrieval, and display.

These four classes of commands were specified through the use of the Action Analyzer, which was fully coded and debugged in the third quarter of 1970. The Analyzer is a program that was designed specifically to aid in producing a comprehensive and unambiguous set of commands for using the on-line SPIRES II system. Command statements are written as they would appear to a user. Then they are expressed in modified Backus Naur Form (BNF), a formal metalanguage used to describe the form of other languages. The equation-like BNF statements are called "productions." A command to set the case for data input, expressed in BNF, might look like this:

```
<case>::=upper-lower
        |upper
        |uplow
        |lower
```

Reading "|" as "or," this says that the case can be specified by any of the phrases on the right side of the production. Commands written in BNF are input to the Action Analyzer; the Analyzer produces diagnostics indicating errors or ambiguities and describes the type of ambiguity in each production. When all the errors have been corrected, the Analyzer produces an Action list that ultimately will drive the SPIRES II parser. This Action list is a coded representation of the BNF productions and their relations to each other.

The SPIRES II parser was implemented as one component of the SPIRES II subprocessor operating under ORVYL. In conjunction with the Action list, the parser scans commands from terminals, identifies their component parts, and calls the appropriate semantic modules to perform the on-line functions required: search, update, display, etc. The semantic modules, in turn, do this by calling file services routines that operate directly on goal records and access records in user files.

The four components of the SPIRES II subprocessor are the Action list, the parser, the semantic modules, and the ORVYL interface. The interface consists of a series of assembler language routines that permit the subprocessor to use various services of the time-sharing monitor. These routines were designed and implemented in the first quarter of 1971.

On the basis of evaluations completed in the fourth quarter of 1970, the decision was made to write on-line system modules in PL360 and to write application programs in PL/1. PL360 is a variant of 360 assembler language that is designed especially for applications requiring systems programming. PL360 gives the programmer complete control of machine functions such as register loading and also provides some of the logical capabilities of higher-level languages, such as looping and "if...then" statements. PL360 code is as efficient to use as assembler language, and its similarities to higher-level languages make writing and correcting programs easier.

The parser was rewritten in PL360, and several tests were run, parsing command language strings for six simultaneous terminal users. These runs indicated that core usage was reduced from 40,000 characters to 4,000.

Parsing rules in BNF were prepared for the entire SPIRES II command language in the second quarter of 1971. These rules in effect define the vocabulary of the command language, i.e., what is a legal command.

By the end of this reporting period, a first version of all the file services routines had been defined, coded, and put into operation. First versions of three of the on-line semantic modules--search and retrieval, update and maintenance, and display--had been defined, coded, and checked out, and were in operation with a small test file of skeleton records.

The fourth segment of the SPIRES II on-line command language, file definition, is handled by the file definition processor. File definition commands will be used by a "file manager"--i.e., a person responsible for establishing and maintaining a file of data. Through the commands, the file manager provides the system with such information as the name of the file he is creating; the types of users who have access to the file; the operations they may perform (read only, read and modify, etc.); the types of search requests that may be entered; the types of information to be stored in the file; and the types of information to be retrieved. Forms were designed to permit file managers to describe input formats, data elements, internal record contents, and output formats. By the end of the reporting period, a prototype version of implementation of a final version had been half completed. The file definition processor (or "file characteristics processor") also had been coded and checked out.

Two major documents were written in the course of the year. The first, Requirements for SPIRES II, gives complete information on using the four areas of the on-line command language described above. The contents pages for this document are attached as Appendix B. The second, Design of SPIRES II, Volume I, describes the design for implementing SPIRES II in the Campus Facility computer. It is attached as Appendix C. Design covers the computing environment, design goals, the structure of the ORVYL interface and the SPIRES II subprocessor, file structure and record structure concepts, and system support functions. Volume II is being prepared; it will give further details of design.

This work in the areas of design, documentation, programming, and testing was carried out in conjunction with (and in part as a result of) the developments described in section 3.1, The Computing Environment. The decision to implement SPIRES II on the Campus Facility computer; the study of interfacing requirements for the Campus Facility software; the analysis of video terminals leading to the selection of the Sanders model; these were all part of the past year's effort.

3.3 Capabilities and Services Planned

The following is a brief summary of the planned scope of the SPIRES II system, as evolved over the past year. The system is here defined in terms of its software. Actual progress made in developing this software, and further detail on some aspects of SPIRES II design, are given in section 3.2, Development Status.

The basic "building blocks" of SPIRES II are a series of standard FILE SERVICES ROUTINES. These operate directly on indexes, records, and data within records. For example, they locate particular key values in indexes, locate particular data element values in records, and transform records from internal to external format for user display and modification. File services will also generate necessary information for recovery procedures.

The SEARCH SEMANTICS MODULES (18 in number), service the on-line command FIND, which is used to enter search criteria interactively. Intermediate search results are ANDed or ORed with previous results, and qualifiers are applied. (A preliminary version was implemented first, in advance of the file definition processor. This version has not, of course, been able to use the file characteristics output of the definition processor. Instead, it used hand-coded characteristics tables. As the file definition processor is completed, these modules will be augmented to interface with its output. This will result in the final version of the search semantics.)

The ten UPDATE SEMANTICS MODULES transform an update transaction entered on-line into external format and add the new or updated record to the deferred update queue. (As with the search semantics, a preliminary version has been implemented first, using hand-coded file characteristics tables.)

In creating, for example, a locally keyed file of data, the externally readable information must be put in a format that makes it accessible to the file services programs. The BATCH BUILD PROGRAM is the batch analog of the update semantics modules.

The DEFERRED UPDATE PROGRAM places the entries in the deferred update queue into the file, building indexes and transforming elements as guided by file characteristics.

The FILE DEFINITION PROCESSOR permits the user to define his files individually, using data element parameters such as length, occurence, and content, and then to store this definition. He may also define user access, input formats, output formats, validity checks, and usage statistics. The user may modify his file definition at a later time if his requirements change.

Rapid recovery from system failure is essential whether the error originated with the user, the software, or the hardware. The recovery must be completed with the user's data in the same condition it was in at the time of failure. In the event of loss, data recently placed in the system must be recoverable. This requires RECOVERY ROUTINES for full and partial data base dumps and for full and partial data base restoration. In addition, programs must be written to reconstruct indexes, data set directories, and the available space table, should these be destroyed.

DISPLAY SEMANTICS MODULES service the TYPE, OUTPUT, and DISPLAY commands. They transform search results to an external format and place them on the device specified in the command.

There remain some thirty MISCELLANEOUS SEMANTICS MODULES that service ancillary commands such as TO SPIRES, EXPLAIN, EXAMPLE, etc. Also in this category are modules to gather usage and data statistics and place them on disk storage at logoff time.

To carry out "housekeeping" functions that ensure efficient daily operation, various UTILITY PROGRAMS are required. These include a data set allocator, a disk space mapper, a file validator, and a program to list file records in external format. A data set allocator organizes data sets for the maximum utilization of space. A disk mapper displays the organization of data sets on disk. A file validator makes routine error checks on files and produces diagnostics. A data base list program organizes the output from a data base dump so that it is easily readable. In addition, system-monitoring routines will record and report types and frequencies of errors and collect user's suggestions.

DOCUMENTATION for both development and production is necessary. The Documents, Requirements for SPIRES II <10> and Design of SPIRES II <9> present, first from the user's, then from the programmer's point of view, the system design being implemented. For production, the system recovery, emergency, and routine operating procedures will be written up for the computer operators. SPIRES programs will be documented so that they can be maintained and modified. Manuals will be provided for users showing them how to create, maintain, and interrogate files. Types of services and costs will be described so a user can select services to meet his needs within the limits of his budget.

3.4 Work in the Coming Year

In the reporting year 1971-72, the SPIRES staff expects to initiate, continue or complete work in seven areas.

1. The file definition processor. The final version of the file definition processor should be completed around the beginning of October 1971. This version will provide for the automatic building of access records and for the generation of record and build characteristics and also search and pass characteristics (see Design for SPIRES II). The format of the final version the processor differs from the first version in that the final version will take processing rules into account. This includes rules applied at the time of input, at the time a search command is issued, and at the time of output or display.

2. The processing rules. By the end of the 1970-71 reporting year approximately 130 processing rules of several types had been defined. Some rules serve double or triple duty

and apply to updating, searching, and passing. Output processing rules generally serve only one purpose. By the end of the coming reporting year all of the processing rules necessary in the SPIRES II system will have been defined.

3. The On-Line Semantic Modules. The final versions of the search and retrieval, update and maintenance, and display semantic modules will be implemented. The master terminal semantics and remaining miscellaneous semantics will be put into operation. Work necessary to implement the parameter commands -- e.g., CROSS, EXTRACT, etc. -- will be completed.

4. Documentation. Volumes II and III of Design for SPIRES II will be published. By the Time the SPIRES II system comes up -- the beginning of June 1971 -- a user's manual will be ready. All the documentation should be completed by the time the system is operational.

5. Software Optimization. Efforts will be made to optimize the on-line system before putting it into production. A study will be made of what ordering of various modules in the on-line system will run optimally in a paging environment. Two of the tools to accomplish this are a sum hardware monitor that will ascertain where execution time is being spent in the on-line processor and a capability called "VITAMINS". The latter is a virtual terminal system for which a script can be written that, when input, simulates many SPIRES users at many terminals, each inputting the same script for several different runs, each time changing parameters or reordering modules in SPIRES, we can evaluate the throughput of the system without worrying about whether or not the loads are the same.

6. Statistics Gathering. In the months to come, it will be determined what kinds of statistics are wanted on the system. Semantic routines will be built to gather statistics at the user interface on such subjects as; what kinds of commands are being used; what kinds of disk accesses are being caused by different commands; what kinds of mistakes are users making; etc. These statistics will help in determining how to alter the user interface so that the SPIRES system is more convenient to use.

7. Policy and Procedures for Operating the System. Procedures will have to be written up for the operators of SPIRES II, so that they will know such things as: how to analyze failure; how to bring up the system from a cold start; what action to take in the case of a crash; how to bring up the system from a warm start. Procedures will also have to be written up for the people in user services. For example, telling how to bring up a file that a user wants to define, and outlining ways to get user feedback on problems with the system. All these procedures will have to be carefully thought-out. Many of them

will involve policy decisions. (For example, what will the pricing algorithm be for overnight processing?)

Decisions must also be made regarding the support of large public files, such as Chemical Abstracts, Nuclear Science Abstracts, etc. Such decisions will be based upon subscription and storage costs, demand, and available funds. Smaller, locally shared and private files will be accommodated on a direct charge-back basis to the file users. They may also, at the file owner's option, be made partially or fully accessible by the user community at large.

4.0 SPIRES AND BALLOTS

Project BALLOTS is a production on-line and batch system being developed to apply a time-sharing computer to bibliographic management <19, 20> in a network of academic libraries in the San Francisco Bay area. The network is called CLAN, the California Library Automation Network, and presently involves Stanford and four other colleges and universities. More members are expected.

The library automation system is expected to provide efficient library technical processing and circulation of library materials, as well as to promote the sharing of resources among the network libraries. The bibliographic data to be used for ordering, cataloging, and so on will be derived partly from the Library of Congress MARC Distribution Services and partly from the network libraries' own input. Ultimately BALLOTS will have four on-line files: a MARC file of 6 to 12 months of the most recent data; an In Process File of all the titles in technical processing; a Catalog Data File of all titles cataloged; and a Circulation Inventory File of all the titles in Stanford's Meyer Undergraduate Library collection. Video terminals will be placed in the various libraries for use by library staff and eventually patrons, and the printed outputs -- purchase orders, catalog cards, spine labels, etc. -- will be produced overnight at the Stanford Computation Center.

SPIRES and BALLOTS have been closely related since their inception <8, 15>. When the first set of BALLOTS services (the BALLOTS -MARC module) is implemented at Stanford in the spring of 1972, it will use much of the software described in this report. Ten more modules, each increasing the range of BALLOTS, will be implemented in the next two years. Generally speaking, the relationship between the two systems, SPIRES and BALLOTS, is the same as what BALLOTS would have had with IMS (Information Management System), GIS (General Information System), or any other preexisting software it might have chosen as a given for development. Simply stated, the general-purpose SPIRES II software provides an environment for the BALLOTS application. BALLOTS programmers will augment and alter portions of SPIRES software, and combine it with BALLOTS software (particularly batch application programs).

APPENDIX A

(REFERENCES)

SPIRES
A BIBLIOGRAPHY OF REPORTS AND PUBLICATIONS

This list does not include quarterly progress reports submitted to the National Science Foundation or presentations before local professional associations.

ED numbered documents are available from LEASCO Information Products, Inc., P. O. Drawer O, Bethesda, Maryland 20014.

PB numbered documents are available from the National Technical Information Service (NTIS), Operations Division, Springfield, Virginia 22151.

I PUBLICATIONS

1. Addis, Louise. "SLAC Library Monitors Underground Physics Press." THE SLAC NEWS, no. 3 (June 2, 1971), pp. 2-3.

2. Martin, Thomas H., and Edwin B. Parker. "Designing for User Acceptance of an Interactive Bibliographic Search Facility." Paper prepared for discussion at the invitational workshop, "The User Interface for Interactive Search of Bibliographic Data Bases," sponsored by the AFIPS Information Systems Committee, January 14-15, 1971, at Palo Alto, California. To be published in the workshop Proceedings (AFIPS Press).

3. Parker, Edwin B. "Behavioral Research in Development of a Computer-Based Information System." In Nelson, Carnot E., and Donald K. Pollock, eds., COMMUNICATION AMONG SCIENTISTS AND ENGINEERS. Lexington, Mass: D. C. Heath & Co., 1970. Pp. 281-92.

4. Parker, Edwin B. "Democracy and Information Processing." EDUCOM (Bulletin of the Inter-University Communications Council), no. 5 (1970), pp. 2-6.

5. Parker, Edwin B. "Developing a Campus Based Information Retrieval System." In PROCEEDINGS, STANFORD CONFERENCE ON COLLABORATIVE LIBRARY SYSTEMS DEVELOPMENT (at Stanford University, Stanford, California, October 4-5, 1968). Stanford University Libraries, Stanford, California, 1969. Pp. 213-30. <ERIC document number ED 031 281; microfiche $ .65, hard copy $11.50.>

6. Parker, Edwin B. "Information Utilities and Mass Communication." In Nie, N., and H. Sackman, eds., INFORMATION UTILITIES AND SOCIAL CHOICE. Montvale, New Jersey: AFIPS Press, 1970. Pp. 51-70.

II DOCUMENTS AND REPORTS

7. Ferguson, Douglas, ed. PROJECT CONTROL NOTEBOOK. 2nd ed., rev. SPIRES/BALLOTS Project, Stanford University, Stanford, California, December 1970. 180 pp.

8. Ferguson, Douglas. "Information Retrieval (SPIRES) and Library Automation (BALLOTS) at Stanford University." SPIRES/BALLOTS Project, Stanford University, Stanford, California, November 1970. 12 pp. <ERIC document number ED 008 543; microfiche $.65, hard copy $3.29.>

9. DESIGN OF THE STANFORD PUBLIC INFORMATION RETRIEVAL SYSTEM (SPIRES II). SPIRES/BALLOTS Project, Stanford University, Stanford, California, July 1971. 385 pp.

10. REQUIREMENTS FOR SPIRES II. SPIRES /BALLOTS Project, Stanford University, Stanford, California, April 1971. 58 pp. <ERIC document number ED 048 747; microfiche $.65, hard copy $3.29.>

11. SPIRES 1967 ANNUAL REPORT. SPIRES/BALLOTS Project, Stanford University, Stanford, California, December 1967. 58 pp. <ERIC document number ED 617 294; microfiche $.65, hard copy $3.29.>

12. SPIRES 1968 ANNUAL REPORT. SPIRES/BALLOTS Project, Stanford University, Stanford, California, January 1968. 111 pp. <NTIS document number PB184 960; microfiche $.95, hard copy $3.00.>

13. SPIRES 1969-70 ANNUAL REPORT. SPIRES/BALLOTS Project, Stanford University, Stanford, California, June 1970. 129 pp. <ERIC document number ED 042 481. The 1969-70 Report is available without charge from the SPIRES/ BALLOTS Documentation Office, Cypress Annex, Stanford, California 94305.>

14. SPIRES REFERENCE MANUAL. 2nd ed., rev. SPIRES/BALLOTS Project, Stanford University, Stanford, California, January 1969. 63 pp.

15. SYSTEM SCOPE FOR LIBRARY AUTOMATION AND GENERALIZED INFORMATION STORAGE AND RETRIEVAL AT STANFORD UNIVERSITY. SPIRES/BALLOTS Project, Stanford University, Stanford, California, 1970. <ERIC document number ED 038 153. Available from the SPIRES/BALLOTS Documentation Office, Cypress Hall Annex, Stanford, California 94305, for $7.50 prepaid.>

III MAJOR PRESENTATIONS

16. Epstein, A. H. "Information Flow." American Society for Information Science, Los Angeles Chapter, Los Angeles, California, December 1970.

17. Epstein, A. H. "Information Flow Analysis." Canadian Society for Information Science, Ottawa, Canada, March 1971.

18. Parker, Edwin B. "A System Theory Analysis of Feedback Mechanisms for Information Systems." Paper read at the FID International Congress of Documentation, September 21-24, 1970, at Buenos Aires, Argentina.

IV ARTICLES ABOUT SPIRES/BALLOTS

19. "Libraries Seek University-Wide Computer Information Service." CAMPUS REPORT, January 14, 1970, p. 7.

20. "Library Goal: Computerized Information Retrieval System." STANFORD OBSERVER, January 1970.

21. A report on automation plans (SPIRES and BALLOTS) at Stanford. COLLEGE AND RESEARCH LIBRARIES NEWS, March 1970, p. 83.

22. Review of the system scope document. COLLEGE AND RESEARCH LIBRARIES, May 1971, pp. 236-38.

V FILM

23. SPIRES/BALLOTS REPORT. Department of Communications, Stanford University, Stanford, California, 1969. <A 15-minute, color, 16mm film giving an overview of the library automation and information retrieval problem in general and of Stanford's approach to it. Written and directed by D. B. Jones. Copies may be rented from Extension Media Center, University of California, Berkeley 94720. Rental charge, $15.00 for 24 hours; purchase price, $180.00.>

APPENDIX B

Requirements for SPIRES II

Table of Contents

REQUIREMENTS
FOR
SPIRES II

AN EXTERNAL SPECIFICATION FOR THE
STANFORD PUBLIC INFORMATION RETRIEVAL SYSTEM

SPIRES
a project of
the National Science Foundation
Edwin B. Parker, Principal Investigator

APRIL 1971

SPIRES/BALLOTS PROJECT
STANFORD UNIVERSITY
STANFORD, CALIFORNIA
94305

CONTENTS

Page

ACKNOWLEDGMENTS 1
INTRODUCTION 2

Chapter

1.0 THE SPIRES USER

- 1.1 User Audience 1-1
- 1.2 The User Consultant 1-1
- 1.3 General Comments 1-2

2.0 THE SPIRES SYSTEM

- 2.1 File Definition 2-1
- 2.2 File Maintenance and Update 2-1
- 2.3 Search and Retrieval 2-1
- 2.4 Display 2-1

3.0 SOFTWARE, FACILITIES, AND CONCEPTS

- 3.1 The SPIRES Environment 3-1
  - 3.1.1 MILTEN 3-1
  - 3.1.2 WYLBUR 3-1
  - 3.1.3 STSM 3-1
- 3.2 The SPIRES Facilities 3-3
  - 3.2.1 The 2741 Terminal 3-3
  - 3.2.2 The CRT Terminal 3-4
  - 3.2.3 SPIRES System Commands 3-5
- 3.3 STSM System Commands 3-8
- 3.4 Definitions and Concepts 3-8
  - 3.4.1 Data Elements 3-8
  - 3.4.2 Records 3-9
  - 3.4.3 Indexes 3-13
  - 3.4.4 Files 3-14

4.0 FILE DEFINITION

- 4.1 Design 4-1
- 4.2 A Typical Session 4-1

5.0 FILE MAINTENANCE AND UPDATE

- 5.1 Concepts and Definitions 5-1
  - 5.1.1 General 5-1
  - 5.1.2 External Data Format 5-1
- 5.2 On-Line Entry and Deferred Update .. 5-3
- 5.3 On-Line File Management Commands .. 5-3
  - 5.3.1 Update Execution Commands 5-3
  - 5.3.2 Record and Data Element Commands 5-4

5.4 Batch Services 5-7
5.4.1 Batch Update 5-7
5.4.2 Batch Conversions 5-8
5.4.3 Batch Utilities 5-8

6.0 SEARCHING AND RETRIEVING

6.1 General 6-1
6.2 Search Language 6-2
6.2.1 Browse Commands 6-2
6.2.2 Retrieval Commands 6-4
6.2.3 Batch Searching 6-10

7.0 DISPLAYING OUTPUT

7.1 Output Formats 7-1
7.1.1 System Default Format 7-1
7.1.2 Formats Defined at File Definition Time 7-1
7.1.3 Formats Defined During Session Time 7-1
7.2 Output Language 7-2
7.2.1 Type Command 7-2
7.2.2 Output Command 7-3
7.2.3 CRT Commands 7-4

APPENDIX C

Design of SPIRES II, Volume 1

This material, written for use by SPIRES programmers, is subject to continual revision as the design of SPIRES II progresses. Anyone wishing to use or quote this material should first contact John Schroeder, SPIRES Systems Development Group Leader, for information about its current state.

DESIGN OF THE STANFORD PUBLIC INFORMATION RETRIEVAL SYSTEM

(SPIRES II)

Volume I

July 1971

SPIRES/BALLOTS Project

Stanford University, Stanford, California

PREFACE AND ACKNOWLEDGMENTS

This document is a technical paper describing the general design and structure of the SPIRES II on-line processor and its batch support programs. We assume that the reader is familiar with time-sharing concepts and vocabulary, and that he has some acquaintance with the software system at the Campus Facility of the Stanford University Computation Center and with the document REQUIREMENTS FOR SPIRES II (SPIRES/BALLOTS Project, Stanford University, Stanford, California, 1971).

The research and development necessary for this paper were carried out by the members of the Information Systems Group of the Campus Facility, Stanford Computation Center (SCC). They are:

Richard Guertin
William Kiefer
Herbert Ludwig
Thomas Martin
John Schroeder (Group Leader)
Cheryl Stevens
Jerrold West

Our appreciation is extended to T. David Phillips, Deputy Director, SCC; James Moore, Group Manager, Campus Facility; Richard Levitt, Group Leader, Campus Facility Systems; John Borgelt, Campus Facility Systems; and James Powell, Campus Facility Systems. The successful implementation of SPIRES II up to this point would not have been possible without their encouragement and support.

Special thanks are due to Jennifer Hartzell, who edited the draft document.

CONTENTS

Introduction I-1

Chapter 1 Environment, Goals, and Components

1.1 ENVIRONMENT 1-1

1.1.1 Existing Campus Facility Hardware 1-1
1.1.2 Hardware Additions for SPIRES II 1-1
1.1.3 Existing Campus Facility Software 1-3
1.1.4 Advantages to SPIRES II of the Campus Facility Environment 1-7
1.1.5 Modifications Required in the Campus Facility System 1-8
1.1.6 The WYLBUR Interface 1-9
1.1.7 The Batch Interface 1-10
1.1.8 The Manual Interface 1-10

1.2 GENERAL DESIGN GOALS 1-11

1.2.1 Openness to Modification 1-11
1.2.2 Ease of Use 1-11
1.2.3 Generality 1-12
1.2.4 Structural Simplicity 1-13
1.2.5 Modularity 1-13
1.2.6 Reliability and Recovery 1-13

1.3 GENERAL COMPONENTS OF DESIGN AND IMPLEMENTATION 1-13

1.3.1 Interspersion of Syntax and Semantics . . . 1-15
1.3.2 Use of ACTION BNF as a Flow Diagram 1-16
1.3.3 The Generalized Parser 1-16

1.4 IMPLEMENTATION CONVENTIONS 1-18

1.4.1 Compiler Languages for Implementation . . . 1-18
1.4.2 The Use of Standard Routines and Structures 1-19

Chapter 2 SPIRES Software Structures

2.1 ORVYL INTERFACE ROUTINES 2-1

2.1.1 INIT 2-1
2.1.2 QUIT 2-1
2.1.3 GETCLOK 2-1
2.1.4 GETCORE 2-3

2.1.5 RETCORE 2-3
2.1.6 SLEEP 2-3
2.1.7 ATCHF 2-3
2.1.8 ATCHNEW 2-3
2.1.9 SCRATCH 2-3
2.1.10 DTCHF 2-4
2.1.11 READF and WRITEF 2-4
2.1.12 RESERVE 2-4
2.1.13 RELEASE 2-4
2.1.14 TPUT 2-4
2.1.15 TGET 2-4
2.1.16 TPROMPT 2-4
2.1.17 WYLBUR 2-5
2.1.18 WILSEN and WYLCON 2-5
2.1.19 READTXT 2-5
2.1.20 GETCOMZ 2-5
2.1.21 GETCOMZL 2-5
2.1.11 PUTCOMZ 2-5

2.2 ACTION LIST 2-5

2.2.1 Elements of ACTION BNF 2-6
2.2.2 The ACTION List Format 2-12

2.3 THE PARSER 2-16

2.3.1 General Description 2-16
2.3.2 Changing Input Levels 2-18
2.3.3 Calling Semantic Modules 2-19
2.3.4 Input Bounding 2-19

2.4 SNAP 2-19

2.5 SEMANT 2-20

2.5.1 Declarations 2-20
2.5.2 Local Procedures 2-20
2.5.3 Group Router 2-22
2.5.4 Semantic Process Router 2-22
2.5.5 Semantic Process Group 2-22

2.6 THE SAVE STACK 2-22

2.7 THE MASTER TABLE 2-24

2.7.1 TSAVER 2-24
2.7.2 TI 2-24
2.7.3 DI 2-24
2.7.4 NXTSPACE 2-24
2.7.5 FLAGS 2-24
2.7.6 USACCT 2-24
2.7.7 USPSWD 2-24
2.7.8 960-Byte Reserved Area 2-25

2.8 THE PARSER TABLE 2-25

2.9 THE FLOW OF CONTROL 2-28

2.9.1 Entry to SPIRES II 2-28
2.9.2 INITIAL 2-28
2.9.3 Call to SEMO 2-28
2.9.4 Call to the Parser 2-28
2.9.5 Calls to Semantic Processes 2-28
2.9.6 Call to the ORVYL Interface 2-28
2.9.7 QUIT 2-28
2.9.8 Branch to SNAP 2-28

Chapter 3 Logical File Concepts

3.1 INTRODUCTION 3-1

3.2 FILE SYSTEM DESIGN REQUIREMENTS 3-1

3.3 FILE STRUCTURE OVERVIEW 3-2

3.3.1 Record Types 3-3
3.3.2 Pointers 3-3
3.3.3 Goal Records. 3-3
3.3.4 Access Records. 3-3
3.3.5 Record Contents 3-6
3.3.6 Passing Data Element Values 3-7

3.4 PROFILES 3-7

3.5 THE HIERARCHICAL STRUCTURING OF DATA ELEMENTS 3-8

3.6 FILE CHARACTERISTICS 3-8

Chapter 4 Organization of Data Sets for Access and Storage

4.1 INTRODUCTION 4-1

4.2 THE ORVYL ENVIRONMENT 4-1

4.2.1 Data Management Under ORVYL 4-1
4.2.2 Contiguous Data Sets 4-1
4.2.3 Accessing ORVYL Files 4-2

4.3 FILE SETS AND DATA SETS. 4-2

4.3.1 File Sets 4-2
4.3.2 Data Set Naming Conventions 4-2

4.4 THE ORGANIZATION OF RECn DATA SETS 4-3

4.4.1 Slot-Structured Data Sets 4-3
4.4.2 Tree-Structured Data Sets 4-3
4.4.3 Tree Rebalancing 4-5

4.5 DATA REMOVAL 4-10

4.5.1 Definition and Criteria 4-10
4.5.2 The Logical Effects of Removal 4-11

4.6 RECORD SPLITTING 4-11

4.7 A SIMPLE ILLUSTRATION OF PASSING AND REMOVAL 4-14

Chapter 5 Physical Formats

5.1 INTRODUCTION AND DEFINITIONS 5-1

5.2 RECORD FORMATS 5-1

5.3 FILE BLOCK FORMAT 5-2

5.3.1 The Tree Data Set Block Format 5-4
5.3.2 The Non-Tree Data Set Block Format 5-9

5.4 RESIDUAL BLOCK FORMATS 5-9

5.4.1 The Available Space Table 5-11
5.4.2 Supplemental Write Blocks 5-13
5.4.3 Status Information Block 5-13

5.5 THE ACCOUNT NUMBER TREE 5-13

5.5.1 Class Privileges 5-14
5.5.2 Sharing Profiles Among Accounts 5-14
5.5.3 The Format of the Account Number Record 5-14
5.5.4 The Organization of the Account Number Tree 5-17

5.6 THE USER MASTER DATA SET FORMAT 5-17

5.6.1 The Record Characteristics 5-19
5.6.2 The Build Characteristics 5-19
5.6.3 The Search Characteristics 5-22

Chapter 6 Implementation of the SPIRES II Access Method

6.1 INTRODUCTION 6-1

6.2 TASK-ORIENTED SUBROUTINE GROUPS 6-1

6.2.1 SRCHREC 6-1
6.2.2 ADDREC 6-2
6.2.3 DELREC 6-2
6.2.4 RPLREC 6-2
6.2.5 ATCHFILE 6-2
6.2.6 DTCHFILE 6-2

6.3 BASIC FILE SERVICES SUBROUTINES 6-2

6.3.1 Data Set Attaching and Detaching Subroutines 6-7
6.3.2 Node Manipulation Subroutines 6-7
6.3.3 Entry Manipulation Subroutines 6-7
6.3.4 Data Element Access Subroutines 6-8
6.3.5 Input and Output Subroutines 6-9
6.3.6 Data Set Lockout 6-9
6.3.7 Allocation Functions 6-9
6.3.8 Miscellaneous Subroutines 6-10

Chapter 7 SPIRES System Support Functions

7.1 INTRODUCTION 7-1

7.2 THE THREE OPERATING CATEGORIES 7-1

7.2.1 Master Commands 7-1
7.2.2 O/S Batch Commands 7-3
7.2.3 ORVYL User Programs 7-3

7.3 THE FILE MAINTENANCE FUNCTIONS 7-3

7.3.1 BATBUILD 7-3
7.3.2 DEFUPDT 7-4

7.4 ERROR DIAGNOSTIC ROUTINES 7-4

7.4.1 VALIDATE 7-4
7.4.2 FILELIST 7-5

7.5 RESTART AND RECOVERY 7-5

7.5.1 DSZAP 7-5
7.5.2 FULDUMP 7-5
7.5.3 FULRES 7-6
7.5.4 RECOVER 7-6
7.5.5 WARMSTRT 7-6
7.5.6 PASSREC 7-7
7.5.7 DISABLE and ENABLE 7-7
7.5.8 AVSPREC 7-7
7.5.9 MESSAGE 7-7

7.5.10 INHIBIT 7-7
7.5.11 KILL 7-8
7.5.12 MAGIC WORD 7-8

7.6 AIDS TO SYSTEM ADMINISTRATION 7-8

7.6.1 TREREBAL 7-8
7.6.2 DISKMAP 7-8
7.6.3 STAT 7-8

Appendices

A Model 67 Scope Support for SPIRES Project A-2

B Basic Operation of the Stanford Time-Sharing Monitor (ORVYL) A-17

C ORVYL User's Guide A-23

D Scope Support in ORVYL A-124

E ACTION Controlled Translation: A New Approach to BNF A-134

F Linkage Conventions for PL360 . . . . A-153

G Coding and Description Standards for PL360 . . . . . A-155

H Standard SPIRES II Dummy Sections A-162

I ACTION BNF Grammar, SPIRES II Command Language . . . A-172

J ACTION List Macros, SPIRES II Command Language . . . A-184

K PL360 Predefined and SPIRES II Functions A-192

L Removal Tradeoff Table A-193

M SPIRES II File Services Procedures A-197

N Error Codes Returned from ORVINTF in R1 . . . . A-260

LIST OF FIGURES

1. Campus Facility Hardware Configuration 1-2
2. Campus Facility Memory Partitions 1-4
3. Subprocessor-User Relationship 1-6
4. SPIRES System Make-up 1-17
5. Storage Layout, SPIRES II On-Line Subprocessor 2-2
6. ACTION List Production 2-7
7. An Example of ACTION BNF 2-10
8. The ACTION List 2-13
9. Table of Correspondences in ACTION Lists 2-15
10. Parser Production Stack 2-17
11. SNAP--ACTION Parser Trace 2-21
12. An Example of Shared Semantic Processes 2-23
13. SPIRES II File Structure 3-4
14. Relationships Between Record Types 3-5
15. Data Element Structures in External Format 3-9
16. Slot-Structured Data Set 4-4
17. An Example of a Tree-Structured Data Set 4-6
18. Sample Tree 4-7
19. Sample Tree After Rebalancing 4-7
20. Sample Tree After Intense Local Growth 4-9
21. Sample Tree with Well-Distributed Growth 4-9
22. File Set Without Removal 4-12
23. File Set with Removal 4-12
24. An Example of Passing and Removal 4-13
25. Control Information Appended to Various Data Element Values 5-3

26. File Block Formats 5-6

27. File Block Structures 5-7

28. Residual Data Set Organization 5-10

29. The Available Space Mechanism 5-12

30. Venn Diagram of Account Numbers and Psuedo-Account Numbers 5-15

31. Account Number Record 5-16

32. Organization of the Master Data Set 5-18

33. Record Characteristics 5-20

34. Build Characteristics 5-21

35. Search Characteristics 5-23

36. SRCHREC Subroutine Hierarchy 6-3

37. ADDREC Subroutine Hierarchy 6-4

38. DELREC Subroutine Hierarchy 6-5

39. Record Replacement Subroutine Hierarchy 6-6

40. Utility Support for SPIRES II 7-2

INTRODUCTION

PURPOSE

This document was written to serve as a system programmers' guide within the SPIRES II project. It reflects the project's state of development as of May 1971. As the document now stands, it serves as a means of project communication and control. In a year from now, the document will have evolved into a sufficiently accurate, organized, and detailed work to serve as an aid in maintaining the implemented system.

FUTURE CHANGES

It is certain that there will be many additions to the document during the next 12 months. It is likely that some of the material presented here will be changed. At least two kinds of updates are certain:

the issuance of Volume II, and

the issuance of revised editions of chapters whenever changes warrant.

CONTENTS OF VOLUMES I AND II

Volume I describes the project's environment, the development methodology, and the basic software concepts and components. Volume II will describe in detail the design of the semantic routines and support modules, and will enable the reader to study the design and implementation of the individual commands described in REQUIREMENTS.

CHAPTER 1

ENVIRONMENT, GOALS, AND COMPONENTS

1.1 ENVIRONMENT

The SPIRES II system will operate as integral system software of the Campus Facility, Stanford Computation Center. A major portion of the SPIRES software will be used by Project BALLOTS staff in implementing library automation at Stanford.

1.1.1 Existing Campus Facility Hardware

The Campus Facility machine configuration is shown in Figure 1 on the following page. It centers around a one-million-byte IBM 360 Model 67 central processing unit, with high-speed drums for operating system residence and virtual memory; 2314 direct access facilities for medium-speed storage; seven-track and nine-track magnetic tape drives; and appropriate unit record peripherals. This system supports approximately ninety terminals concurrently out of the two hundred IBM 2741 typewriter terminals located on campus and nearby. Other features of the configuration are a PDP-9 linked to the system via a 2701 data adapter and the multiplexor channel, which supports foreign computers, graphics devices, etc.

1.1.2 Hardware Additions for SPIRES II

The shaded components in Figure 1 represent the additions to Campus Facility hardware for SPIRES II.

1.1.2.1 2319 Disk Storage Drives. When the SPIRES II system becomes available to the campus community, there will be nine 2319 drives in addition to the present storage devices. Additional hardware orders will be placed in advance to allow for the addition of four more drives every six months. Each drive will provide storage space for approximately 28 million characters.

1.1.2.2 A PDP-11 Front-end Computer. All communications between SPIRES II and cathode ray tube (CRT) terminals will be via a PDP-11 "front-end" machine. Appendix A contains a discussion of the PDP-11 design and hardware alternatives involved. Also listed is a bill of materials covering all items required for the terminal - PDP-11 communications links.

1.1.2.3 Terminal Equipment. In addition to the 2741 typewriter terminals, SPIRES II will be available via two types of CRT terminals--an inexpensive upper-case terminal (not yet chosen) and a more expensive upper/lower-case terminal (Sanders Associates 800 series). Appendix A gives the hardware configuration for terminal equipment in greater detail. It is

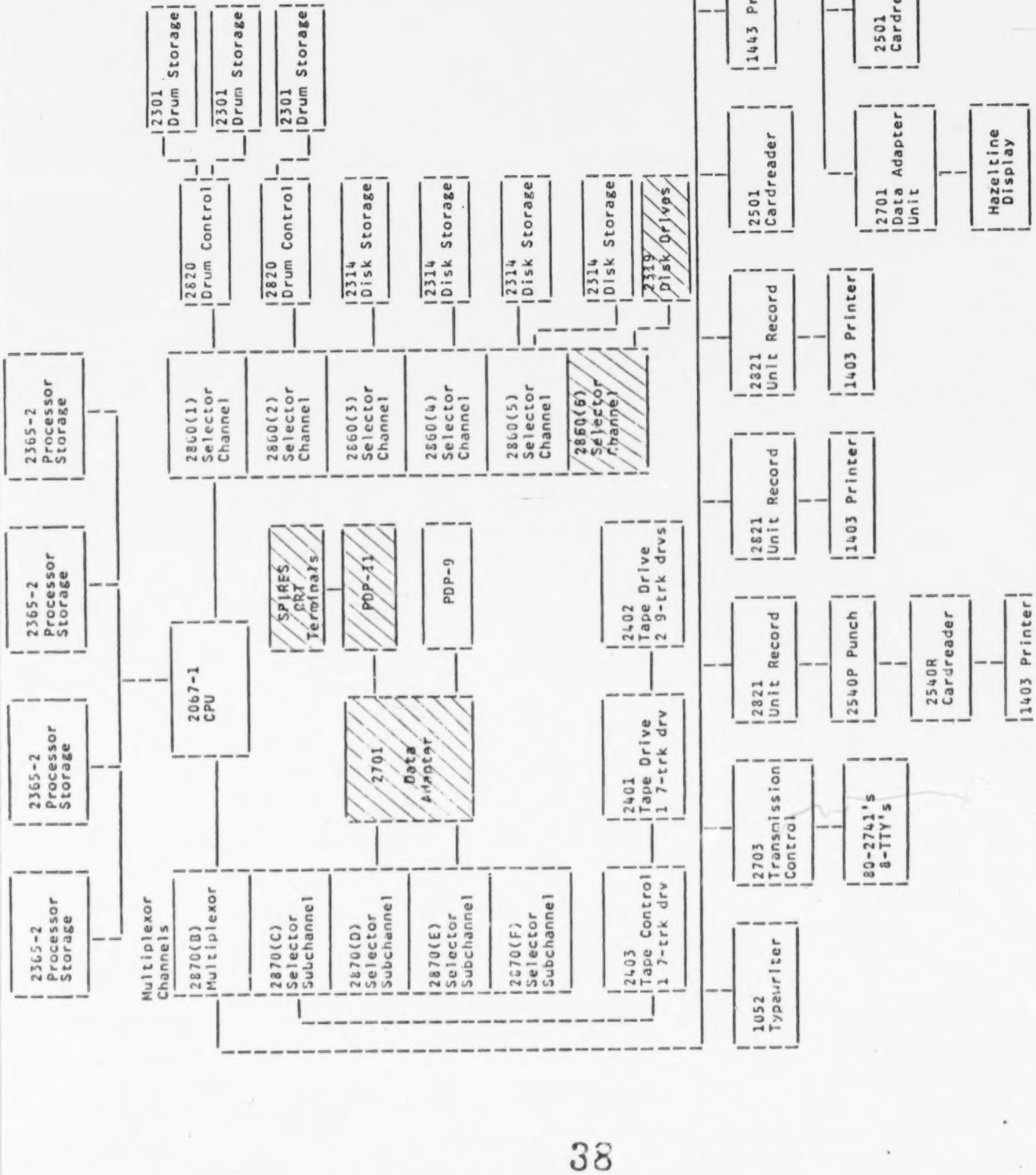

2365-2 Processor Storage
2365-2 Processor Storage
2365-2 Processor Storage
2365-2 Processor Storage
2067-1 CPU
Multiplexor Channels
2870(B) Multiplexor
2870(C) Selector Subchannel
2870(D) Selector Subchannel
2870(E) Selector Subchannel
2870(F) Selector Subchannel
2701 Data Adapter
SPIRES CRT Terminals
PDP-11
PDP-9
2860(1) Selector Channel
2860(2) Selector Channel
2860(3) Selector Channel
2860(4) Selector Channel
2860(5) Selector Channel
2860(6) Selector Channel
2820 Drum Control
2820 Drum Control
2301 Drum Storage
2301 Drum Storage
2301 Drum Storage
2314 Disk Storage
2314 Disk Storage
2314 Disk Storage
2314 Disk Storage
2319 Disk Drives
2403 Tape Control 1 7-trk drv
2401 Tape Drive 1 7-trk drv
2402 Tape Drive 2 9-trk drvs
1052 Typewriter
2703 Transmission Control
80-2741's 8-TTY's
2821 Unit Record
2540P Punch
2540R Cardreader
1403 Printer
2821 Unit Record
1403 Printer
2821 Unit Record
1403 Printer
2501 Cardreader
1443 Printer
2701 Data Adapter Unit
Hazeltine Display
2501 Cardreader

Figure 1. Campus Facility Hardware Configuration

(Shaded areas denote additions necessary for SPIRES II.)

estimated that the proposed configuration will be able to support up to forty CRT terminals concurrently (any mixture of the two types); if that number is exceeded, a second PDP-11 can be added to the configuration. Such an addition is unlikely in the foreseeable future.

1.1.3 Existing Campus Facility Software

The Campus Facility 360/67 provides a broad range of computational services, including text editing, remote job entry, batch compilation and execution, interactive compilation and execution (time sharing), and specialized partitions for short jobs and utility programs. Three basic assumptions guided the development of the system.

> The various parts of the system should interact closely and complement one another.
>
> Time sharing should be limited to those uses of the system requiring it.
>
> The overall system should be optimized continually so as to execute the job load with maximal efficiency.

The following describes the major partitions of the Campus Facility System. Reference may be made to Figure 2.

1.1.3.1 Operating System. This is presently release 18.6 of OS/360, MFT-II. It is conceivable that a shift could be made to MVT in the future; this in no way affects the design of SPIRES II.

1.1.3.2 High-Speed Batch/FUTIL. This partition runs in two modes. (1) On first and second shifts, jobs of short duration are run. (2) On third shift, file utility runs (IEHMOVE, etc.) are run. In each case, a small partition monitor exercises control to restrict execution to certain language processors and utilities. In the case of high-speed jobs, the monitor also performs the function of the O/S Job Scheduler. Input and ouput handled in the high-speed programs are limited to unit-record peripherals and scratch data sets on disk. By substituting limited but optimized functions in High-Speed Batch for those of the operating system, the average job time in this partition has been reduced to two seconds. The language processors supported in this partition are SPASM (a single-pass assembler), WATFIV, LISP, BASIC, ALGOLW, XALGOLW, PLC, and any O/S load modules that meet the I/O requirements. ALGOLW accounts for the bulk of the jobs run in this partition.

1.1.3.3 Large Batch. This partition is available to users of FORTRAN-G, LISP, GPSS, FORTRAN-H, the G-level Assembler, PL/1, COBOL, PL360, and others.

SCC Campus Facility
360/67 Core Storage

Partition	Size
O/S NUCLEUS	92,000 bytes
HIGH-SPEED BATCH	132,000 bytes
LARGE BATCH	276,000 bytes
ORVYL	222,000 bytes
WYLBUR	88,000 bytes
MILTEN	66,000 bytes
O/S WRITER	10,000 bytes
HASP	138,000 bytes

Figure 2. Campus Facility Memory Partitions

1.1.3.4 Stanford Time-Sharing Monitor (ORVYL). This partition is the one in which SPIRES II will execute. It is the only one in the system that uses the time-sharing hardware peculiar to the model 67. The monitor itself resides (unpaged) in 120,000 bytes of the partition. The remainder of the partition is divided into 35 4,096-byte pages.

Programs reside on a 2301 drum, and are also segmented into 4,096-byte pages. The drum pages are termed "virtual memory": only as they are required for execution must any of these pages reside in real memory. If an executing program calls for a page of itself not presently in real memory, it is interrupted and relieved of control until the needed page has been transferred from virtual to real memory. Because a program page may execute in many locations in real memory, the address operands within the program page conform to the address space of virtual, not real, memory. It is necessary to translate these virtual addresses to real addresses as instructions are fetched and effective addresses generated. This process is done in the hardware, and is known as "dynamic address translation." (See Appendix B for a more detailed explanation.)

The programs executed under ORVYL are divided into two categories: subprocessors and user programs. Suprocessors are bodies of code written by professional system programmers that serve many users simultaneously. (see Figure 3). Current examples are the Interactive LISP Processor and Stanford BASIC. An outstanding design characteristic of subprocessors is re-entrancy. A subprocessor never stores within itself; each user attached to a particular subprocessor has his own work area, which is logically appended to the subprocessor code whenever it executes in his behalf. This area may be modified by the subprocessor at will. ORVYL thus allows the designer to consider a subprocessor as a single program divided into two parts: an unmodifiable part (code) and workspace. The designer need not consider the fact that there will be a number of concurrent users, since ORVYL appends the appropriate user workspace and restarts the code for the user at whatever point it may have been interrupted. SPIRES II will execute as a subprocessor.

User programs need not be re-entrant, and may be written only in FORTRAN, ALP, or PL360. Generally, they may only interact with only one terminal. However, external devices such as graphics scopes, foreign computers, paper tape I/O devices, and so on, may also be attached.

A restriction common to both subprocessors and user programs is that they may not issue OS supervisor calls. All requests for supervisor services must be made via a special subset of ORVYL supervisor calls. The required parameter setup and the actual supervisor calls are accomplished by coding from a special set of ORVYL macros. These macros are fully enumerated and described in Appendix C, the User's Guide for ORVYL.

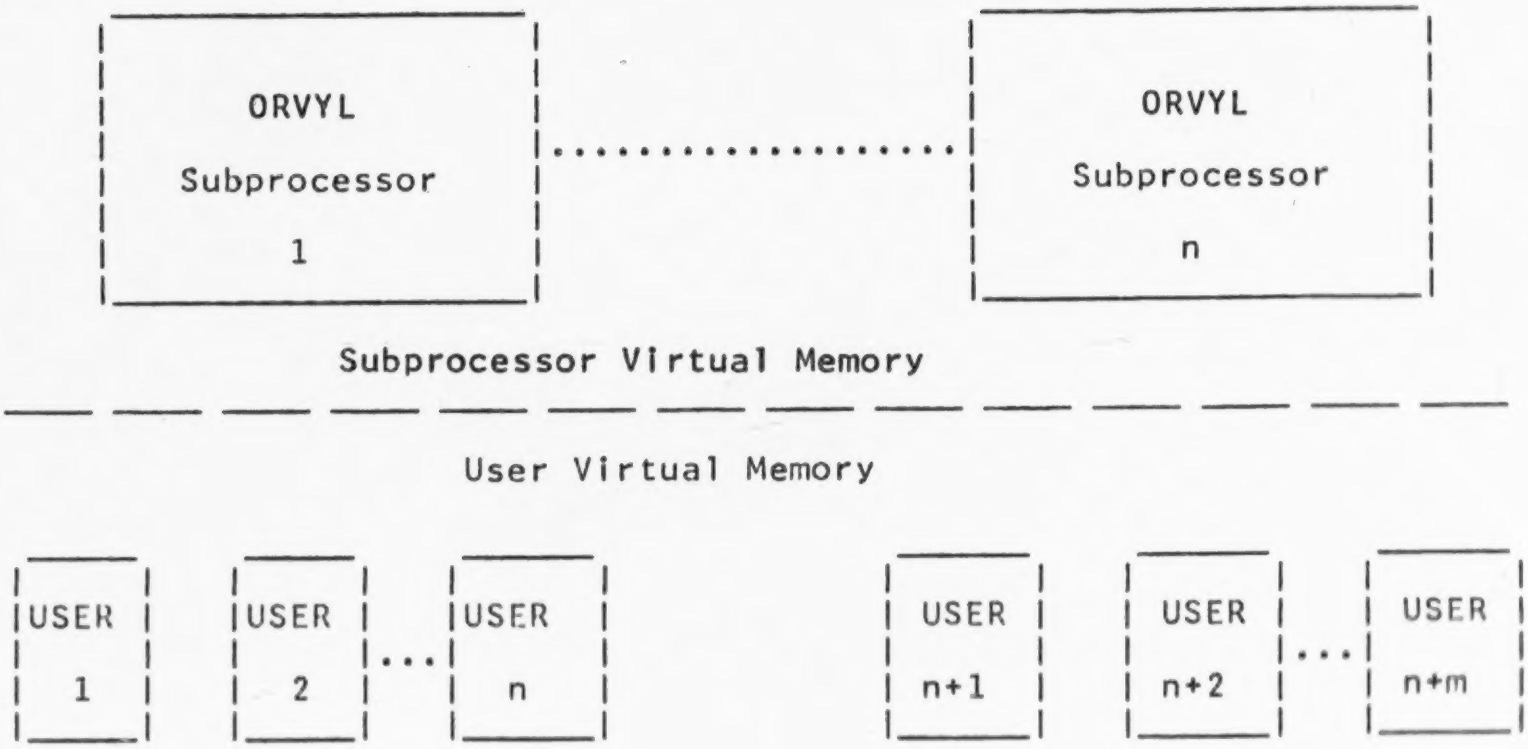

ORVYL
Subprocessor
1
ORVYL
Subprocessor
n
Subprocessor Virtual Memory
User Virtual Memory
USER
1
USER
2
USER
n
USER
n+1
USER
n+2
USER
n+m

Figure 3. Subprocessor-User Relationship

A disk file capability is provided as part of the ORVYL macro set. To OS, the ORVYL disk file set appears to be one data set extending over multiple disk packs. ORVYL subdivides the space into 2,048-byte blocks, and user files consist of non-contiguous collections of these blocks. Associated with each user file is one drum-resident directory entry, one disk-resident Master Index Record (MIXR), and an indeterminate number of disk-resident Secondary Index Records (SIXR's) that map user logical record numbers into physical record numbers. These files may be connected with a user by a subprocessor, or attached to a user program.

1.1.3.5 WYLBUR. This partition contains the Stanford Text Editor. A 2741 terminal user may call in an OS disk data set; manipulate it using line-level or character-level editing commands; replace the old copy on disk; submit a copy to the batch job stream for execution; retrieve batch execution results; and display them at his terminals. New data sets may be created by copying from other data sets, or by collecting (keying) data line by line at the terminal. Programs to be run interactively under ORVYL (i.e. user programs) are submitted through WYLBUR.

1.1.3.6 MILTEN. MILTEN is the terminal communications processor. Currently, it can service up to eighty 2741's and eight teletypes at one time. The system allows any six of the terminals to be dial-up; the rest come in via leased lines. It works on an interrupt basis, assigning a buffer from a pool to each terminal logged on to the system. The buffer and its associated control blocks are termed Remote Terminal Blocks (RTB's) and are the basis for interpartition communication. If, for example, a user is connected with WYLBUR, he has two RTB's assigned to him--one in MILTEN and one in WYLBUR, with data transfer via a "MOVE CHARACTER" instruction. The same situation exists for users connected with an ORVYL subprocessor.

1.1.3.7 HASP. HASP provides spooling of the unit record input and output to and from the batch partitions. It also interfaces with the remote job entry commands in WYLBUR, and with the commands that enable a user to fetch batch execution results at a terminal.

1.1.4 Advantages to SPIRES II of the Campus Facility Environment

SPIRES operating as a subprocessor under ORVYL yields many advantages:

> ECONOMY. The user pays only for time during which the subprocessor serves him.
>
> EXISTING SOFTWARE. The ORVYL time-sharing software and the 2741 terminal software are in operation now. SPIRES and WYLBUR can be interfaced so closely that the dividing line will not be immediately apparent to the user. This is discussed further below.

QUICK RESPONSE TIME. Preliminary studies have shown response time to be on the order of 1.5 to 2 seconds for simple searches.

EASE OF DEVELOPMENT. ORVYL has a complete set of on-line, interactive debugging aids that will materially assist the implementation of SPIRES II.

The main alternative to developing SPIRES II as an ORVYL subprocessor is to develop or to find an already existing swapping monitor, and to locate a 200,000-byte partition somewhere at Stanford in which SPIRES could reside. During early and mid-1970, it was considered economically feasible to develop an independent data facility at Stanford. But this approach was later ruled out owing to a constricting financial situation. Exhaustive studies were made of every possible installation at the University, with negative results.

If such a partition were suddenly to be available, IBM's Time-Sharing Option (TSO) seems the most viable of existing alternative packages. It has not yet been released, however, and to commit ourselves to such a package would require hands-on experimentation and study over a period of several months. From a cursory study we feel that TSO would not equal ORVYL with respect to either response time or ease of development. It should be noted that a paging system using the hardware features of the 360/67, especially if optimized (ORVYL), should be far superior in performance to a swapping system that depends entirely on software (TSO). The decision has therefore been taken to implement SPIRES II as a subprocessor under ORVYL.

1.1.5 Modifications Required in the Campus Facility System

Before the project decided to implement SPIRES II as an ORVYL subprocessor, it was recognized that modifications to the Campus Facility System would be necessary. Stanford Computation Center has made the necessary expertise available at no cost to the project to accomplish these modifications, which are described below.

1.1.5.1 ORVYL File System. The present file system under ORVYL provides excellent support for small user files, which are transient in nature and are easily re-created if a system failure occurs. But the requirement that a table lookup be done (to reach Master and Secondary Index Records--MIXR's and SIXR's) each time a data block is called for doubles the input-output load on a system handling large files. Although the fact that the records in any one file are not contiguous optimizes the use of disk space among all users, it also doubles the number of seeks required to use one record. Furthermore, the impossibility of isolating a file in one spot makes recovery from failure needlessly complex.

An extension will therefore be provided to the ORVYL file system to permit files to be declared and maintained as physically contiguous collections of records. The need for MIXR's and SIXR'S with contiguous files will be eliminated, allowing the indexes necessary for quick entry into SPIRES files to exist as the top level of the access tree.

1.1.5.2 Subprocessor Communications Area. An ORVYL subprocessor, as mentioned above, is re-entrant, and is written as if only one user existed. Thus the subprocessor, while acting on behalf of one user, may not have access to the workspace of any other user. It is therefore not possible at present for the subprocessor to "remember" anything as it goes from user to user. A resident (unpaged) area, 256-bytes in length, will therefore be provided for each subprocessor in the system. Read and write access to this area will be via extensions to the ORVYL macro set. An additional feature will be "READ WITH LOCKOUT" to prevent simultaneous updating of the area.

1.1.5.3 Virtual Access Method (VAM). This facility will allow programs running in a batch partition to access ORVYL disk data sets, either contiguous or non-contiguous. The facilities for file access will be identical externally. This means that the same macros and call sequences will be used, thus allowing on-line file access modules to be used in batch mode with no source code modification. The batch programs may be run whether ORVYL is executing at the time or not. The only restriction (an insignificant one) is that no more than one such batch program may be executing in the system at one time.

1.1.5.4 CRT Terminal Small Computer Support. ORVYL and MILTEN must be modified to allow CRT terminal support via a front-end PDP-11. Appendix A discusses the alternatives considered prior to this design decision. It also gives the cost and scheduling involved, and Appendix D contains the proposed scope support addendum to the ORVYL User's Guide.

1.1.6 The WYLBUR Interface

ORVYL macros exist to pass commands to WYLBUR for execution, to read the contents of a user's working data set (the data set he is currently editing), or to write into the data set. In this manner, a user attached to SPIRES may enter WYLBUR collect mode by issuing an ADD or SUBSTITUTE command to SPIRES; he may then type in his file update, edit it, and issue another command to cause SPIRES to read the contents of the working data set and apply them to the user's file. In general, SPIRES will pass any command that it cannot recognize to WYLBUR. (If WYLBUR doesn't recognize the command either, it passes it back with an error return code.)

1.1.7 The Batch Interface

1.1.7.1 System-Provided. The SPIRES user may choose not to enter his file updates on-line. Instead, he may wish to collect a large group of updates using WYLBUR, save the group in an OS data set, and inform SPIRES of its existence via a BATCH UPDATE command. The system will store the data set name in a special system data set. The batch build program will use this data set in order to find the location of data sets intended for input. The batch build program, running under ORVYL as a user program, will then take the OS input data sets one at a time, using the WYLBUR interface, convert their contents to internal format, and link the results to the batch queue for processing by the deferred update program.

The system will also provide a batch search capability via the BATCH SEARCH command. When SPIRES recognizes this command, the contents of the WYLBUR working data set (assumed to contain the search command) are read and parsed; if no error diagnostics occur, the command is written into a special system data set, which will be used by the batch search processor as a command stream. This processor runs under ORVYL as a user program, and is nearly identical to the search routines in the on-line system.

1.1.7.2 User-Provided. By using the SPIRES command "OUTPUT WYLBUR," the user may transfer search output (sorted if desired) to his WYLBUR working data set. He may then issue the WYLBUR command "SAVE <dsname> on <volume>." WYLBUR will then cause the results to be placed in a sequential disk data set, where they may be reached and manipulated with a batch statistical package of the user's choice.

1.1.8 The Manual Interface

It will be necessary to employ at least one full-time equivalent to perform the task of monitoring SPIRES II and administering it daily. The various functions could be carried out by several part-time persons, or one full-time person. The following is a list of the necessary functions to be performed.

> RECOVERY MANAGEMENT. In case of failure involving loss, damage must be quickly assessed and a method prescribed for making corrections. In complex cases, consultation and assistance from the Information Systems Group will be obtained.
>
> STATISTICAL MONITORING. Statistics will be kept on command usage (user behavior), file content (where allowable), and system component usage. From these statistics, conclusions must be drawn about command language adjustments, file characteristic adjustments, and promising areas for optimization in the subprocessor.

SPACE MANAGEMENT. Since disk space will be limited, file growth rates, the use of multiple extents, the dividing up of available space, and future space requirements will have to be monitored and coordinated. Recommendations to increase the disk storage capacity will originate here.

BATCH JOB SUBMISSION AND MONITORING. Runs such as batch build, deferred update, and so on, must be regularly submitted for execution. The results of these runs must be passed through a quality control check, and corrective action taken where appropriate.

USER CONSULTING. When a prospective user approaches the system for the first time, he should be given the background and materials to get him started. As he measures the system against his requirements, he should be aided in choosing the correct strategies for his data organization and storage. Finally, he should be assisted through the the file definition process. During the time he is using the system, he may encounter problems; these should be brought to the attention of the User Consultant (see section 1.2 of REQUIREMENTS FOR SPIRES II, SPIRES/BALLOTS Project, 1971).

1.2 GENERAL DESIGN GOALS

The subjects discussed below are all goals to be pursued in the General Design phase of SPIRES II development.

1.2.1 Openness to Modification

The SPIRES II command language differs markedly from that of SPIRES I in both design and scope. Therefore, despite the fact that a great deal of experience was gained with SPIRES I, some of the new command language will undoubtedly require modification and improvement. And it must be possible to modify parts of the language without overly affecting the rest of the system. Furthermore, SPIRES II may in all probability be extended during 1972-73 to support types of applications not presently envisioned. The system should be designed in an open-ended fashion so that extensions can be accomplished with as little disturbance to existing code as possible.

1.2.2 Ease of Use

In addition to such obvious aids as choosing simple command verbs, the following points should be kept in mind as ways to reduce the amount of information a user must remember.

1.2.2.1 Freedom of Movement. Minimum demands should be made on the user's awareness of where he is in the system or where he may go next. A user should be able to shift from one function to another (e.g., from searching to updating and back again) without explicitly informing the system of these moves.

The SPIRES II command language allows combinations like the one shown below, which illustrates the desired flexibility.

```
(logon sequence)
-? select file preprint
-? find author jones
23 DOCUMENTS FOUND
-? remove record 12345
RECORD 12345 REMOVED
-? and title aardvark
2 DOCUMENTS FOUND
-? type
(output displayed)
```

1.2.2.2 Prompting for Missing Information. Some commands cannot be issued unless a prerequisite command has been given. In those cases where it is possible the system should prompt the user to issue the missing command.

```
(logon sequence)
-?find author jones
-NO FILE SELECTED
SELECT? (user responds to prompt with filename)

(logon sequence)
-?select file preprint
-?and author jones
-AND REQUIRES A PRECEDING FIND COMMAND
```

In the first example, the system was able to prompt for missing information. In the second example, this was not possible, so an error diagnostic was issued and the command ignored.

1.2.2.3 Consistency. Rules should rarely have exceptions. An example of a good rule in SPIRES II is the abbreviation of command verbs--all abbreviations are formed using the first three characters of the command verb.

1.2.2.4 Minimum Prerequisite Knowledge. System users--whether casual searchers, data input clerks, or file managers--should not be required to understand (or use) technically oriented features such as job control language, recovery procedures, etc.

1.2.3 Generality

The design will not be unduly slanted towards storing and retrieving the data peculiar to one discipline rather than another. SPIRES II, like SPIRES I, will handle bibliographic data, but not to the exclusion of clinical data, statistical data, or full text material. Furthermore, the system must be able to encompass extreme cases with minimal loss to overall efficiency and no programmer intervention.

1.2.4 Structural Simplicity

The system should be simple enough in structure to be explainable in a few pages. The relationships between its various components should be easily understood. It should be possible to represent the flow of control in five pages or less, given an understanding of some basic system concepts.

1.2.5 Modularity

Modularity is implicit in the goals stated above, but no design document can be considered complete that does not mention it explicitly.

1.2.6 Reliability and Recoverability

Reliability means that hardware and software errors do not occur during the production execution of the system. Hardware and software errors may never be completely eliminated; we therefore limit ourselves to quick detection of any problems resulting from such errors and prompt recovery of any data lost. The programmatic detection of errors is largely accomplished through redundancy of one kind or another. For example, two sets of redundant data may be used to calculate a result; if the results are different, an error has occurred. Redundancy also plays a role in recovery. If data in one form is lost and it exists in another, the lost data may be regenerated by transformation. The SPIRES II file design makes quick recovery and programmatic validation possible by redundant storage of critical control information.

During the execution of an on-line processor, there are times when the system or a user file is vulnerable. A prime example is the on-line update of a multiply indexed data base. The update sequence commonly involves twenty or more writes. If the sequence is interrupted by a system crash, the file being updated loses its integrity. By routing on-line updates to a batch operation (while simulating an on-line update for the user's benefit), the SPIRES II design has reduced this vulnerability to a point where it can be completely overcome by redundant writing to disk.

1.3 GENERAL COMPONENTS OF DESIGN AND IMPLEMENTATION

The precise definition (and redefinition) of a user command language is a problem of some magnitude. The language must seem natural to the user (so that he can go ahead on instinct if all else fails), but it must also lend itself to simple recognition and breakdown by the system. This process of recognition and decomposition is usually called "parsing."* In order to be parsed, a language must not contain circular definitions; the elements of each of its commands must be distinguishable from each other; and none of its commands can be open to more than one interpretation.

To describe all possible combinations of language elements clearly and concisely is a lengthy if not impossible process. English is an unwieldy and ambiguous tool to use for command language definition. A symbolic metalanguage (a formal language used to describe the syntax of other formal languages) is required to make such definition possible.

The most common metalanguage in use for defining computer languages is BNF (Backus-Naur Form). An example follows:

```
1.    <PROG>::=<paren>
2.    <paren>::=(<expression>)
3.    <expression>::=<non_paren_body>
                    |<non_paren_body><paren>
                    |<paren>
                    |<non_paren_body><body><paren>
                    |<non_paren_body><body>
                    |<body><paren>
                    |<body>
4.    <body>::=<paren><non_paren_body>
               |<body><paren><non_paren_body>
5.    <non_paren_body>::=<non_paren>
                          |<non_paren_body><non_paren>

      .
      .
      .
```

This defines a formal language in which

(A+(B+C)/2)

is allowable. The numbered expressions in the example are called "productions." Production 1 defines a program as composed of one parenthetical expression. Production 2 defines a parenthetical expression as an expression surrounded by parens. Production 3 defines an expression as "(1) a NON_PAREN body, or else (2) a

---

*Computational linguists refer to the rules used to parse command language strings as "productions of a grammar." The language is the set of all possible sentences that can be parsed. Since most programmers do not make that distinction, we will call grammars languages.

NON_PAREN body followed by a parenthetical expression, or else (3) a parenthetical expression...or else (7) a body." During parsing, each time a string of symbols that constitutes a right part is discovered, it is replaced by the left part. For example, a <NON_PAREN_ BODY><PAREN> will be treated as an (EXPRESSION>. A set of productions written in BNF is called a grammar, and is said to describe a formal language.

Over the last three years, McKeeman et al. <1> have developed the XPL system at Stanford. One feature of this system, the XPL Analyzer, takes a grammar written in BNF, diagnoses it for errors, and writes out parsing tables for that grammar. The McKeeman group have written SKELETON, the skeleton of a parser that can be used with the tables. It is called a skeleton since it must be filled out with what are called semantic modules if it is to do anything more than recognize grammatically correct programs. Semantic modules are used to produce machine code or to perform other operations. While the syntax shows how everything fits together, the semantics do the work.

XPL was studied by the SPIRES project during early 1970. The overall approach met with approval: use BNF or something like it to define the command language syntax; then use an analyzer (meta-translator) of some sort to convert the definition into a data structure, which when combined with a general purpose parser would constitute the implemented command language (minus, of course, the semantics).

XPL and BNF, however, were found to have several drawbacks when used to define interactive command languages. These drawbacks were considered serious enough to warrant developing a modified BNF, a new analyzer, and a new parser, each tailored to the problem of implementing an interactive language. Out of this development came the ACTION Analyzer, ACTION BNF, ACTION LISTS, and the SPIRES II parser. Appendix E explains the ACTION approach in contrast to that of XPL.

1.3.1 Interspersion of Syntax and Semantics

When the XPL parsing methodology is used, the parsing tables are proportional in size to the square of the number of terminal symbols. (A terminal symbol is a self-defining term, for example, "(" in <PAREN>). Each time a terminal symbol is added, the skeleton must be changed. With ACTION parsing, the tables are proportional in size to the number of productions. Since our formal language definition tends toward few productions and many terminal symbols, space is thus saved.

With bottom-up parsing, semantics can be performed only when replacing a right-part symbol string with its left-part symbol. A requirement in the SPIRES II system is a close integration of syntax and semantics.

Semantic module numbering is implicit in XPL, and explicit in ACTION. Thus the XPL parser must be compiled when new productions are added, whereas in ACTION, syntax may be changed or new semantics added without having to modify either the parser or the other semantic modules.

The metasymbol

<n>   n = 1,2,3...

is used to number a semantic module explicitly and to indicate the precise point in the parsing process at which it should be executed.

Consider the ACTION BNF production

```
<COMMAND_LANGUAGE>::=<1>COMPILE<ECKS>
                    ||LOGOFF|
                    |SET<SP><SET_CASE>
                    |PAU(SE)<4><9>
                    |<5>(INPUT_LINE)
                    |<INPUT_LINE><3>
          <SE>::=SE
```

Using the underscored right side as an example, one may translate as follows: "As soon as the string 'PAU,' followed optionally by the string 'SE,' is recognized in an input line, call semantic processes 4 and 9." Semantic process 4 checks to be sure that the input line is used up. Semantic process 9 calls the ORVYL interface pause routine.

1.3.2 Use of ACTION BNF as a Flow Diagram

In the above example, one can trace the parsing process as it proceeds from level to level in the language definition. It is also possible, given a list of semantic module definitions in numerical sequence, to determine the actions being performed at any time in the process.

Since semantic processes vary from the obvious to the complex, the more complex ones will require charting in greater detail than the example here contains. The ACTION BNF, then, serves as a first-level flow chart in a two- to three-level hierarchy.

1.3.3 The Generalized Parser

Figure 4 illustrates the process of command language implementation, combining the command laguage with other parts of the system. The parser, which need never be modified, can be combined with any desired ACTION list. This can be done dynamically if desired. There may be multiple ACTION lists within a system, each representing one node in a hierarchy of languages. A semantic routine called by the parser using ACTION

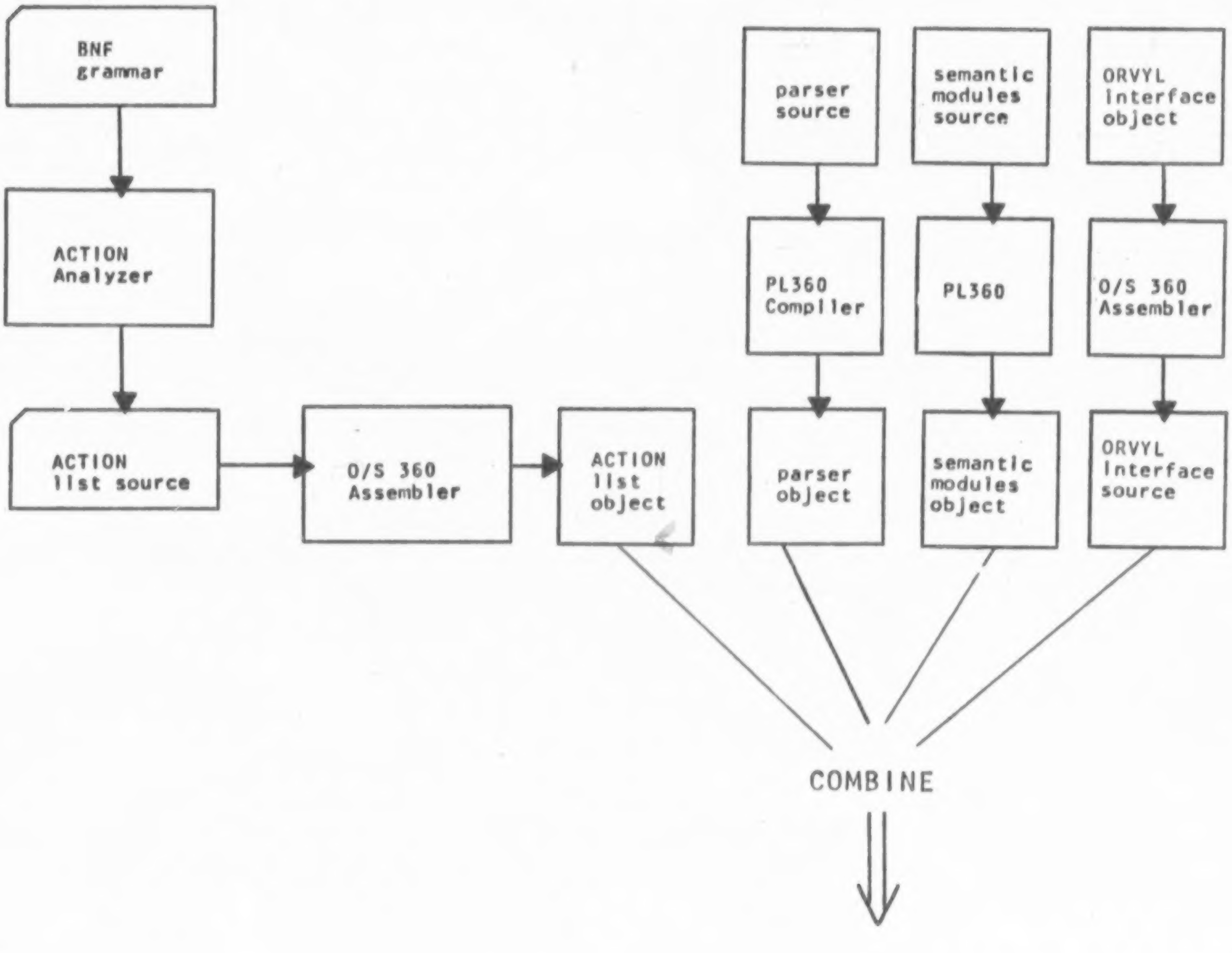

BNF grammar
ACTION Analyzer
ACTION list source
O/S 360 Assembler
ACTION list object
parser source
PL360 Compiler
parser object
semantic modules source
PL360
semantic modules object
ORVYL interface object
O/S 360 Assembler
ORVYL interface source
COMBINE

Figure 4. SPIRES System Make-up

list 1 can initialize the parser to use ACTION list 2, which may in turn cause another semantic routine to start the parser on ACTION list 3, and so on.

The parser is re-entrant, and the ACTION lists are read-only and require no relocation. Therefore, the lists may be readily shared by a number of users, as long as the parser's status (its location within the ACTION list, its position in the input string, and so forth) is stored and maintained for each user so that parsing may be resumed should it be interrupted.

1.4 IMPLEMENTATION CONVENTIONS

1.4.1 Compiler Languages for Implementation

1.4.1.1 Choosing a Language. The development staff of SPIRES II expended considerable effort attempting to find some other implementation language than 360 Assembler Language. Ninety percent of SPIRES I was coded in PL/1. On the basis of that experience, PL/1 has been ruled out as the primary implementation vehicle for SPIRES II. It was found that although PL/1 supported the requisite functions, programs with a high degree of modularity were penalized with considerable overhead. Furthermore, programs written in PL/1 depend on the ability to request supervisory functions directly from OS; they could not be executed under a time-sharing monitor. FORTRAN and COBOL were ruled out on the basis of functions to be supported; it was felt that to use either language to support the data structures found in SPIRES II would be difficult or impossible.

In 1966 Niklaus Wirth had defined and implemented a language called PL360 <2>. This language had, in addition to machine-level functions, ALGOL-like features such as "BEGIN...END," "DO...END," "GOTO...," assignment statements and "IF" statements. The efficiency of the compiled code is equal to that of Assembler Language, and the PL360 compiler is small and fast.

After some test implementations using PL360 (the parser was transliterated from PL/1 to PL360) the language was adopted as a project standard for SPIRES II implementation. The compiler was modified to include an equating capability and a cross-referencing capability, and to compile interactively under ORVYL.

Implementation using PL360 appears to progress at least 100 percent faster than it would using Assembly Language; the code is much more easily read, and may be learned in a week or less by a competent Assembly Language programmer.

The PL360 compiler is now distributed by the SHARE organization and is maintained locally by the Language Support Group of the Stanford Computation Center. Use of the compiler at Stanford is increasing and we suspect that industry-wide use would increase markedly were PL360's value as a system programming language well-known.

1.4.1.2 Language Conventions. All modules or routines that call ORVYL for supervisory services will be written in G-Level Assembler Language (OS/360), using ORVYL macros. These will be gathered together into a multiple-entry-point control section named ORVINTF.

The parser, the dummy sections, and all the semantic routines will be coded in PL360.

The standard linkage conventions between modules are given in Appendix F. Appendix G details the coding conventions for PL360.

1.4.2 The Use of Standard Routines and Structures

To permit naming standardization, standard dummy sections will be used to gain access to elements within system data structures. Any code sequences required in more than one location will be standardized. Appendix H enumerates standard dummy sections.

CHAPTER 2

SPIRES SOFTWARE STRUCTURE

This chapter describes in detail each major component of the SPIRES II subprocessor, except the SPIRES access method. Figure 5 provides an overview, showing each component in storage sequence. The ORVYL interface, the ACTION lists, the PARSER, SNAP, SEMANT, and the DSECTS within user memory will all be dealt with. The last section of this chapter discusses the subprocessor flow of control by describing the numbered arrows in Figure 5. The discussion of the SPIRES access method modules is sufficiently complex and important to warrant devoting a chapter to it (Chapter 5).

2.1 ORVYL INTERFACE ROUTINES

These routines are contained in a multiple entry point control section. Corresponding to each entry point is an ORVYL macro surrounded by interface code; all interfaces with ORVYL are done via calls to one of these routines. Of particular importance are INIT and QUIT, which are the entry and exit routines for the entire subprocessor.

2.1.1 INIT

This routine handles entry to the subprocessor. It locates the beginning of user memory, attaches the user's terminal, sets to upper case, makes the timer and attention routines operable, and initializes the parser. The interface routine "GETCOMZ" is called to bring the 256-byte common area (see section 1.1.5.2). The master terminal and logon flags are checked to make sure the subprocessor is enabled to receive users. If so, semantic process 0 is called and upon return the parser is called, starting with production 1.

2.1.2 QUIT

This routine gains control by being called from a semantic process because of an error condition or because the parser executes a return using R14; QUIT is the label on the next sequential instruction after the BALR in INIT that transferred to the parser at start-up time. A message is emitted, "returned to WYLBUR"; the terminal is detached; and the user is passed back to WYLBUR with his working data set cleared.

2.1.3 GETCLOK

This routine returns the real-time clock value in milliseconds in R1.

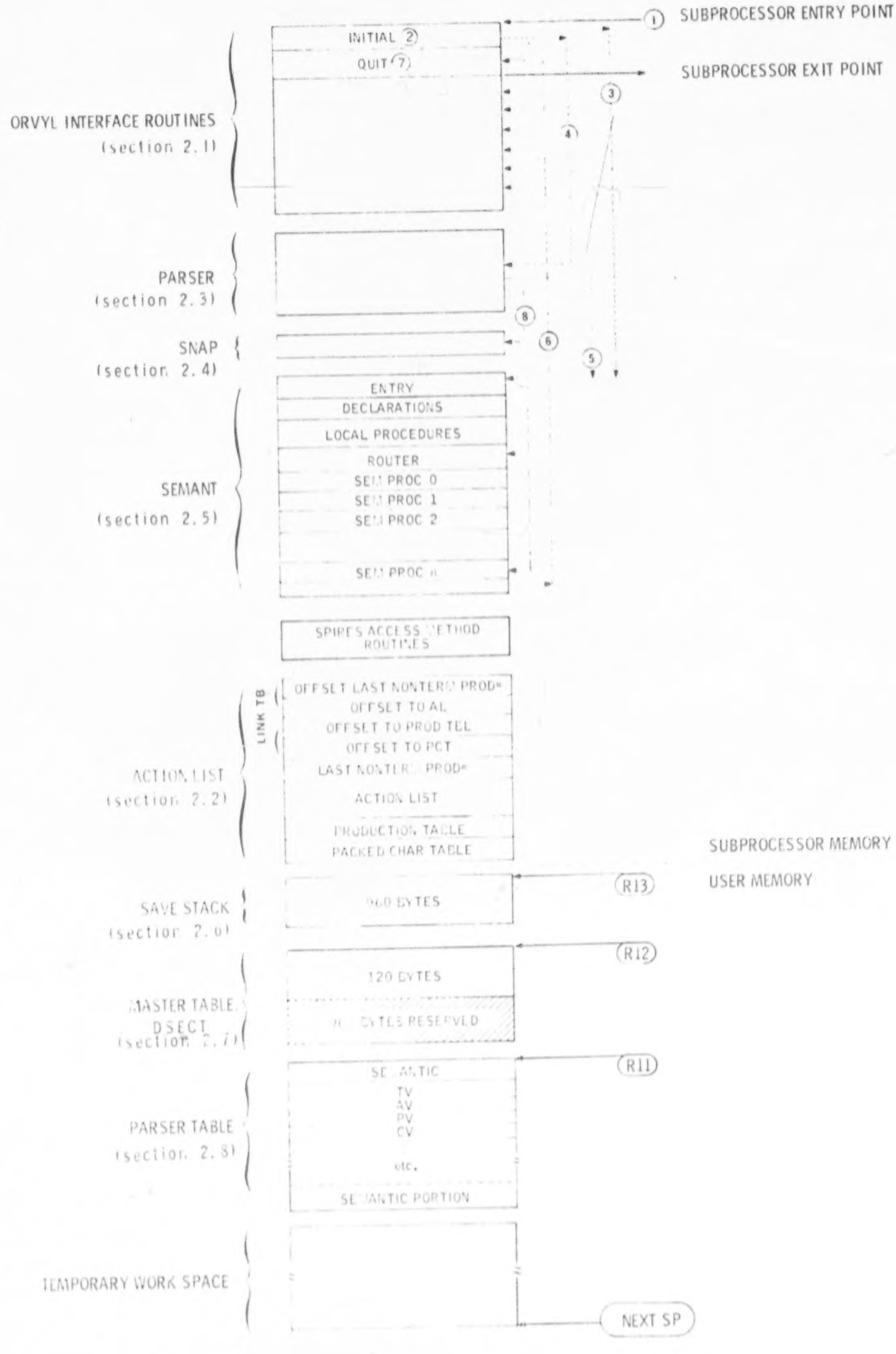

① SUBPROCESSOR ENTRY POINT
INITIAL ②
QUIT ⑦
SUBPROCESSOR EXIT POINT
③
④
ORVYL INTERFACE ROUTINES (section 2.1)
PARSER (section 2.3)
⑧
⑥
⑤
SNAP (section 2.4)
ENTRY
DECLARATIONS
LOCAL PROCEDURES
ROUTER
SEM PROC 0
SEM PROC 1
SEM PROC 2
SEM PROC n
SEMANT (section 2.5)
SPIRES ACCESS METHOD ROUTINES
LINK TB
OFFSET LAST NONTERM PROD#
OFFSET TO AL
OFFSET TO PROD TBL
OFFSET TO PCT
LAST NONTERM PROD#
ACTION LIST
PRODUCTION TABLE
PACKED CHAR TABLE
ACTION LIST (section 2.2)
SUBPROCESSOR MEMORY
USER MEMORY
R13
960 BYTES
SAVE STACK (section 2.6)
R12
120 BYTES
BYTES RESERVED
MASTER TABLE DSECT (section 2.7)
R11
SEMANTIC
TV
AV
PV
CV
etc.
SEMANTIC PORTION
PARSER TABLE (section 2.8)
TEMPORARY WORK SPACE
NEXT SP

Figure 5. Storage Layout, SPIRES II On-Line Subprocessor

2.1.4 GETCORE

This routine keeps track of requests for virtual memory made in behalf of each user. A pointer, NXTSPACE, exists in the master table DSECT (see section 2.7). When this routine is called, R1 contains the number of bytes of core desired. Nxtspace is incremented by that amount, rounded up to the next doubleword boundary, and restored in the table. If a page boundary has been crossed, a load is done from the new page to place it in keep status. When the caller gets control back, R1 points to the core just allocated. If no core is available, R1 contains zero. If R1 contains zero at entry to this routine, R1 is returned pointing to the start of the next area to be allocated.

2.1.5 RETCORE

This routine deallocates core using the reverse of GETCORE. When the routine is called, R1 contains the address beyond which core is to be deallocated. If a page boundary is crossed, a set macro is issued to free that page and make it available to other users.

2.1.6 SLEEP

When this routine is called, R1 contains a count in milliseconds. If R1 is equal to or less than zero, one second is assumed. The user in whose behalf the call was issued is placed in the wait state for the stated interval.

2.1.7 ATCHF

This routine attaches an ORVYL file to a user. R1 contains a pointer to the filename. R2 contains the length of the filename. R3 contains zero or nonzero, according to whether or not the file is old or new. R2 is returned with zero if the attach was successful, and nonzero if it was not. R1 contains the device identifier, which is stored in the user's master table.

2.1.8 ATCHNEW

This routine is used to attach a new file for unshared use. If the file already exists, it is scratched. At entry to this routine, R1 contains a pointer to the filename; R2 contains the length of the filename. At exit, R1 is zero (successful attach) or nonzero (unsuccessful attach).

2.1.9 SCRATCH

This routine deletes an attached file. At entry to this routine, R1 contains the device ID (DI). At exit, R1 is zero or nonzero, depending on success or failure. DI is zeroed.

2.1.10 DTCHF

This routine detaches a file from a particular user. R1 contains the device identifier.

2.1.11 READF and WRITEF

These routines read and write a specified block from and to the file whose identifier is stored in DI. At entry to either routine, R1 contains the block size, R2 contains the record number, and R3 contains the core address where the input or output is to take place. At exit, R1 contains the record size and R2 contains the next block number; or, R1 contains an error code (see the ORVYL Guide) and R2 contains zero.

2.1.12 RESERVE

Reserves a file for exclusive control. The file is determined by the contents of DI.

2.1.13 RELEASE

Releases a file from exclusive control. The file is determined by the contents of DI.

2.1.14 TPUT

This routine causes a message to be written to the user's terminal. At entry to this routine, R1 contains a pointer to the message to be written. R2 contains the length, a number equal to or less than 132. Nothing is returned.

2.1.15 TGET

This routine causes a message to be read from the user's terminal. At entry to this routine, R1 contains a pointer to an area to receive upper-case input and R2 points at an area to receive upper/lower-case input; if R1 and R2 are equal, then upper-case input only is assumed. At exit, R2 contains the length of the message received. It should be noted that if the user types "<blank>...<carriage return>" or "<blank>...<attn>," he is reprompted. If the user just types ATTN, he is placed in collect mode unless he was already in collect mode, in which case he is returned to command mode.

2.1.16 TPROMPT

This routine causes a prompt to be written to the user's terminal, and a message to be read back. At entry to this routine, R1 points to an area to receive upper-case input. R2 points to a prompt message block (aligned on a full-word boundary) that contains a full-word length of prompt, followed by the prompt message. At exit, R2 contains the length of the message received back. It should be noted that if the user types "<blank>...<carriage return>" or "<blank>...<attn>," he is reprompted. If the user just types ATTN, he is placed in collect mode.

2.1.17 WYLBUR

This routine is called whenever it is necessary to cause a WYLBUR command to be executed in the user's behalf. At entry to this routine, R1 points to the command string to be passed to WYLBUR, and R2 contains the length of the string. At exit, R1 will contain zero if the command execution was successful.

2.1.18 WILSEN and WYLCON

These routines issue either SENSE or CONTROL macros to determine the user account number, password, terminal ID, number of lines in the WYLBUR working data set, etc.

2.1.19 READTXT

This routine is used to read text from the WYLBUR working data set. At entry to this routine, R1 points to the area in user core to receive the text, R3 contains length information, and R2 points to a block of control information (aligned on a full-word boundary) that contains a code (see the ORVYL Guide), the number of lines to be read, the first line number, and the last line number.

2.1.20 GETCOMZ

This routine obtains the 256-byte communications area and places it into a designated area of user virtual memory.

2.1.21 GETCOMZL

This routine is identical to the above, except the lockout feature is invoked. This means that the copy being obtained is for update, and other users may not obtain copies until a PUTCOMZ has been executed (see below).

2.1.22 PUTCOMZ

This routine causes a 256-byte area of user virtual memory to replace the current contents of the communication area.

2.2 ACTION LIST

To discuss the ACTION list and its format, it is first necessary to explain ACTION BNF and its elements; the ACTION list is nothing more than a compact list representation of the BNF grammar that defines the SPIRES command language. As indicated in section 1.3.1, the ACTION list is generated by passing an ACTION BNF grammar through the Analyzer (see Figure 6). The Analyzer produces a source deck, composed almost entirely of ACTION macro statements. When the source deck is prefixed by the ACTION macro definition deck and they are assembled, a data structure composed of absolute constants is created. Appendix I contains the ACTION BNF grammar for the SPIRES II command language. Appendix J contains the corresponding ACTION list source.

2.2.1 Elements of ACTION BNF

ACTION BNF is composed of a series of expressions called productions. A series of productions is called a grammar and defines a formal language--in the case of SPIRES II, an interactive, on-line command language.

Productions comprise two parts, the left part and the right part, written:

```
LEFT_PART::=RIGHT_PART
```

There may be alternate right parts, written:

```
LEFT_PART::=RIGHT_PART1
           |RIGHT_PART2
           |RIGHT_PART3
           |RIGHT_PARTn
```

Left parts may contain only one term, which defines the production, whereas right parts may contain several terms:

```
Term 1::=term 2 term 3
        |term 4
        |term 5
Term 2::=term 6
Term 3::=term 7 term 8
```

ACTION BNF can be considered a programming language, with the terms as the language elements. The right-part terms can be thought of as calls to a closed subroutine, and the left-part terms as subroutines names. There are several different types of left parts, and these may be thought of as several categories of subroutines, with a different behavior assigned to each category. There are also several kinds of right-part terms. A central point is whether or not a return to the caller reports success or failure; the difference between various types of calls lies in whether they must succeed or fail, and whether or not they are to be repetitive.

A BNF grammar thus may be thought of as a set of nested subroutine calls or, alternatively, a hierarchy of subroutines.

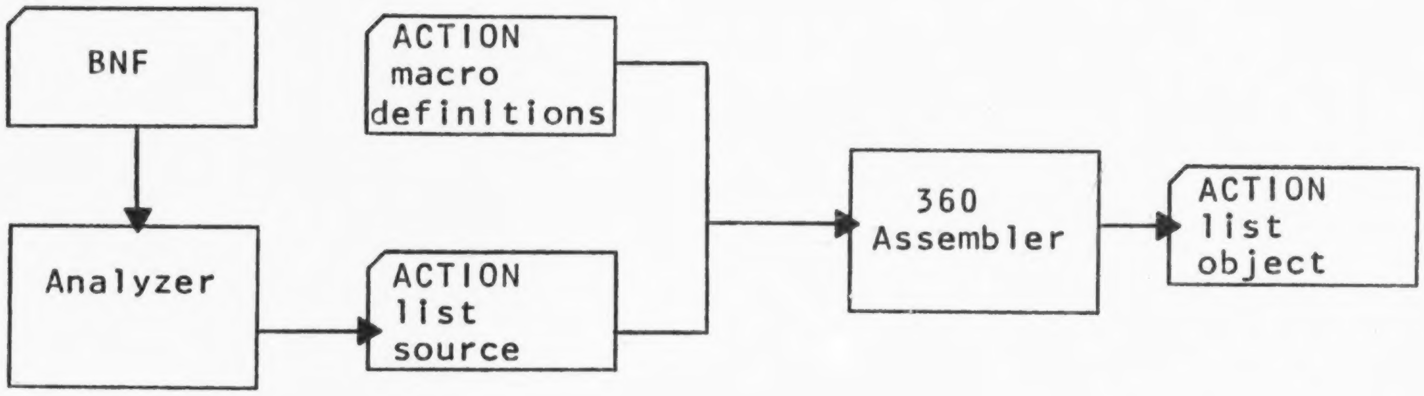

BNF
Analyzer
ACTION macro definitions
ACTION list source
360 Assembler
ACTION list object

Figure 6. ACTION List Production

Often, no action is performed by a "subroutine" save the calling of other "subroutines." Eventually, a call will be made to a subroutine that actually performs some function, such as scanning for legal or illegal characters or making comparisons to find a specified character string.

The following sections discuss the different types of productions (left-part terms) and the different kinds of production calls (right-part terms).

2.2.1.1 Production Types. A grammar written in ACTION BNF is composed of four types of productions. These productions, described below, differ in format and interpretation.

The Comment. The comment may be inserted anywhere in the set of productions except between alternate right parts. In format, this statement simply requires an asterisk in position one of the input record.

The Standard Production. The standard production is identifiable by "<name>" in the left part. "name" must appear in a right part of another production. Standard productions are used to show alternative ways command language elements fit together. During parsing, the first right part is used. If it fails, then the next right part is taken. If none of the alternate right parts succeeds, then the production fails.

The Terminal Production. The Terminal Production is used to define some subset of the 256 eight-bit codes that is allowable in a command element string. A minimum and maximum length must be specified. The terminal production is identifiable by "(name)" in the left part. When a terminal production is called the byte code subsets are reserved in a TRT table, and the input string is scanned for illegal characters. The occurence of such a character stops the scan. The length of the scanned string is then compared with the minimum and maximum lengths to ensure that it falls within the allowable range.

The Basic Production. The basic production is identifiable by "|name|" in the left part. Unlike the standard production requirement, it is not necessary for "name" to appear in a right part of any other productions. Basic productions are used in the same way as standard productions during parsing. There are three kinds of basic production:

> Nonreferenced. In this case, "name" does not appear as a term in any right part of any other production. Every ACTION BNF grammar contains at least one basic nonreferenced production, the goal symbol. Basic nonreferenced productions may only be entered semantically, i.e., by setting the P register to the production number and calling the

parser. This always happens where the user enters the system, because the parser is called from INIT with P=1; hence the goal symbol production is a basic nonreferenced production.

Referenced. In this case, "name" appears as a term in the right part of another production. Using the symbol "|name|" as a left part serves to place the following restriction on the "|name|" production: none of the productions called by the right parts of "|name|" may in turn call "|name|." The following example contains recursion, which contradicts the use of vertical bars in the left part.

```
(example)  |x|::=<y><z>
                .
                .
                .
           <y>::=<x>
```

The use of vertical bars in this case serves as a precaution against inadvertent recursion.

Redefined. In this case, "name" is used as the left part of a previous standard or basic production.

```
(example)  |x|::=<a><b>
                .
                .
                .
           |x|::=<y>

(example)  <x>::=<a><b>
                .
                .
                .
           |x|::=<y>
```

In these two examples, if the right part "<a><b>" parses the input substring successfully, the redefining production is entered to reparse the same input substring. If the redefinition fails, the original right part also fails.

2.2.1.2 Formation of Right Parts. Right parts may be composed of several kinds of terms, with some restrictions as to which terms may occur together, and which terms may be in the right parts of certain types of production.

Required Link. This term is the most common kind of right-part term. In Figure 7, "<BUILD_LANGUAGE>" is a required link. The presence of this term in a right part means, "call the

```
|COMMAND LANGUAGE|::=(0,MASTER_LANGUAGE)<LOGOFF>
 <MASTER_LANGUAGE>::=<BUILD_COMMAND><BUILD_LANGUAGE>
                    | |LOGOFF|
                    | <EXTRA COMMANDS>
  <BUILD_LANGUAGE>::= <1> BUILD (SP) <4>
                    .
                    .
                    .
          <LOGOFF>::= LOG(OFF)
             <OFF>::= OFF
                    .
                    .
                    .
              (SP)::= 0,1,1,40
```

Figure 7. An Example of ACTION BNF

production whose left part is <BUILD_ LANGUAGE>." If the called production fails to parse, the right part of the calling production fails and the next alternate right part is tried. If the required link is in the last alternate right part of the calling production, the calling production fails.

If the called production succeeds, the parser continues with the next term in the right part containing the required link. If there are no other terms, the calling production succeeds.

Lookahead link. This term must appear by itself in a right part, and it cannot be the last alternate right part in a production. In Figure 7, "|LOGOFF|" is a lookahead link. It means, "call the production whose left part is <LOGOFF>." If the <LOGOFF> production reports failure, the right part of the calling production fails. If the <LOGOFF> production reports success, then the entire calling production fails immediately.

Character string. The occurrence of a character string without surrounding brackets, parentheses, or vertical bars in a right part causes the parser to compare a substring of the input line with the character string. In Figure 7, the string "BUILD" is a call to the character scanner. The success or failure of character strings follows the same pattern as that of required links.

Semantic link. The occurrence of "<n>" as a right-part term means, "at this point in the parsing call semantic process n." In Figure 7, the <BUILD_LANGUAGE> production, when called, will call semantic process 1 to read a line from the terminal before it calls the character scanner to look for the string "BUILD."

Optional link, standard. The occurrence of the symbols "(name)" or "(1, name)" (the two symbols are equivalent) in a right part means, "call the production <name>, |name|, or (name)." The parser continues to the next term in the right part containing the standard optional link, regardless of whether the called production succeeds or fails. If there are no other terms, the calling production succeeds. Figure 7 contains uses of the standard optional link: the call for optional spaces, "(SP)," in the <BUILD_LANGUAGE> production and the call for optional occurrence of the string "OFF" in the <LOGOFF> production. The latter case also shows how command abbreviations may be handled.

Optional link, call until failure (CUF). The occurrence of the symbol "(0, name)" as a right-part term means, "keep calling the production <name>, |name|, or (name) until that production fails." Upon failure, the parser continues to the next term in the right part containing the cuflink. If there are no other terms. the calling production succeeds. The term "(0, MASTER-LANGUAGE)" in Figure 7 is a CUFLINK: it calls

"MASTER_LANGUAGE>" until it fails; a call is then made to the <LOGOFF> production.

Optional link, pseudo-recursive. The occurrence of the symbol "(2, name)" indicates that a transfer to the <name> or |name| production should occur, but that the action level is not incremented. This amounts to a transfer from one part of the ACTION list to another without the parser's "remembering" where the transfer came from. Such a call must be the last term of any right part in which it is used.

Class scan terms. Class scan terms can only be right-part terms in terminal productions. The right part of the (SP) production in Figure 7 contains class scan terms, and it may be interpreted, "look for a string, minimum length = 1, no maximum length, consisting of hex - 40's (blanks)." If non-blank strings were to be looked for, a production could be written such as

```
(NON_BLANK)::=0,1,0,40
```

The usual class scan terms are max, min, unlike/like, hexstring, and charstring. Hexstring and charstring are separate entities, so if charstring occurs a comma must follow hexstring, even if hexstring is missing. Permissible combinations are:

```
,hexstring
,hexstring,charstring
,,charstring
```

One example that defines a numeric string not longer than ten characters is

```
(digits)::=10,1,1,,0123456789
```

or, alternatively,

```
(digits)::=10,1,1,F0F1F2F3F4F5F6F7F8F9
```

A non-numeric string could be represented as

```
(alpha_special)::=10,1,0,,0123456789
```

2.2.2 The ACTION List Format

The following is a discussion of the different parts of the ACTION list. Figure 8 shows the layout graphically.

2.2.2.1 LINK#TB. This is a four-element vector containing the offsets, relative to zero, of the other parts of the ACTION list. These offsets must be made permanent before the ACTION list can be used by the parser; this is accomplished by adding

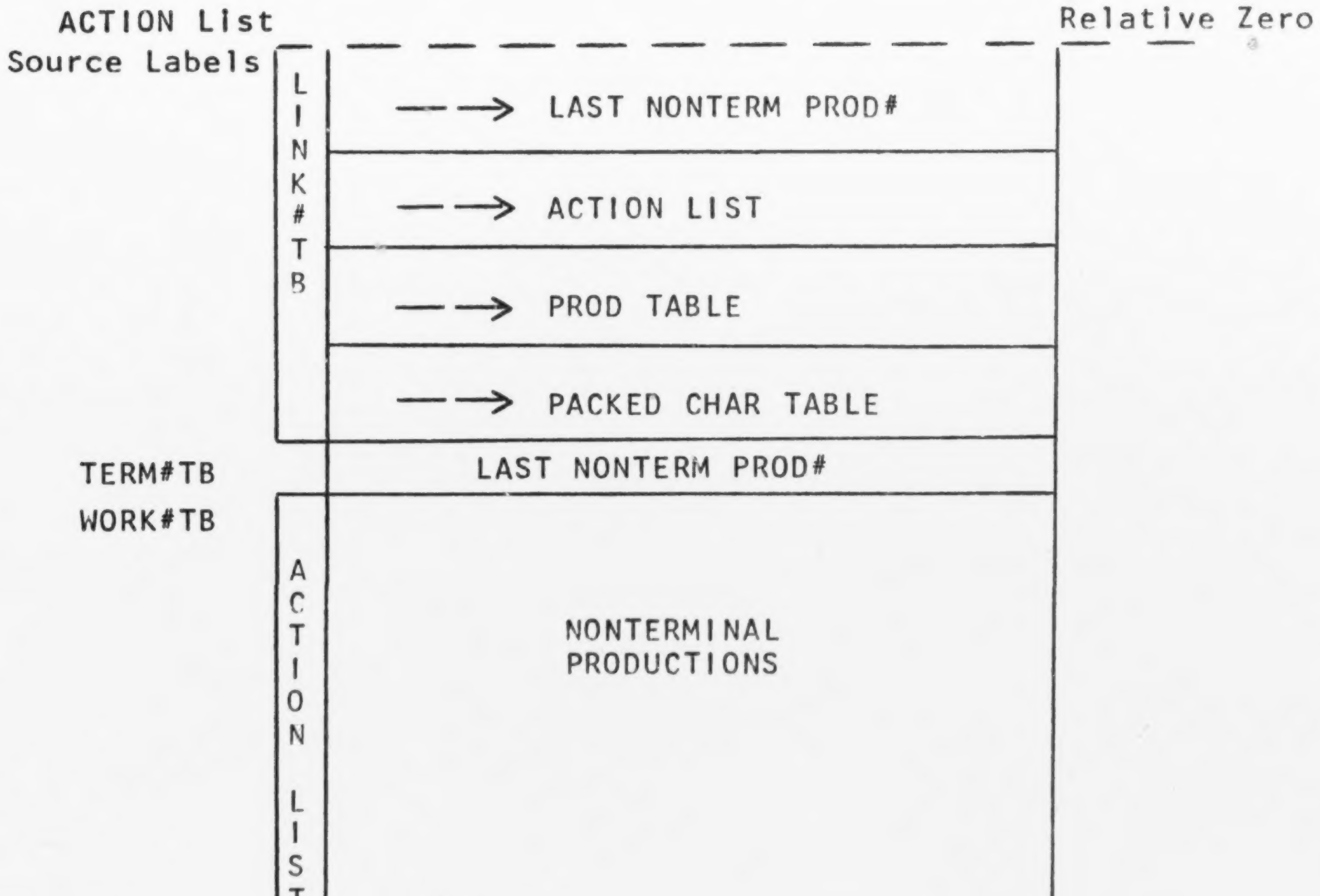

ACTION List
Source Labels
Relative Zero
LINK#TB
--> LAST NONTERM PROD#
--> ACTION LIST
--> PROD TABLE
--> PACKED CHAR TABLE
TERM#TB
LAST NONTERM PROD#
WORK#TB
ACTION LIST
NONTERMINAL PRODUCTIONS
TERMINAL PRODUCTIONS (CLASS SCANS)
PROD#TB
PRODUCTION TABLE
CHAR#TB
PACKED CHARACTER TABLE

Figure 8. The ACTION List

each element of the vector along with the origin address of the ACTION list and storing the result in the parser table.

2.2.2.2 TERM#TB. This is the last nonterminal production number in the ACTION list. (The parser has a separate routine to handle calls to terminal productions. Production numbers are numbered sequentially by the Analyzer, as are right parts.) The parser compares each production number to the value of TERM#TB to ascertain if the production is terminal or nonterminal.

2.2.2.3 WORK#TB. This constitutes the ACTION list proper. Figure 9 shows the correspondence between the BNF left- and right-part terms, the ACTION list source code, and the ACTION list object code. It is the object code that drives the parser.

Each right part of a nonterminal production has the following parts:

```
PROLOG
MACRO1
MACRO2
   .
   .
   .
MACROn
```

where MACRO1...n are any of REQ#LINK, LAH#LINK, RDF#LINK, CHR#SCAN, SEM#LINK, OPT#LINK. If machine code is to be included in an ACTION list, it must be hand-inserted by surrounding the Assembly Language statements with INS#LINK and INS#TERM macros. RDF#LINK is unconventional in that it appears as the last term in each right part of the production being redefined, not of the redefining production itself, and is automatically generated by the Analyzer.

Each right part of a terminal production consists of the macro

```
CLS#SCAN T,X,N,H,L
```

This assembles into a five-byte ACTION list object element. Nonterminal production macros produce either two- or three-byte object elements, with an ACTION code from 1 to 9. CLS#SCAN produces an ACTION code of 0 or 1, depending on whether the string given in the BNF was "like" or "unlike." The "1" with REQ#LINK is not ambiguous because nonterminal productions are handled sparately and distinctly from terminal productions.

2.2.2.4 PROD#TB. This is a table of halfwords, each of which contains the offsets, relative to the beginning of the ACTION list proper, of each production. A production number whose location is to be found is simply doubled and used as an index based on a register containing the address of PROD#TB. The

PRODUCED FROM				ACTION LIST OBJECT				
BNF LEFT PART	BNF RIGHT PART	HAND IN-SERTED	ACTION LIST MACROS	BYTE 1	BYTE 2	BYTE 3	BYTE 4	BYTE 5
	<name>		REQ#LINK P	1	P			
	|name|**		LAH#LINK P	2	P			
|name|*			RDF#LINK P	3	P			
	symbol		CHR#SCAN X, N	4	X	N		
	<s>		SEM#LINK S	5	S			
		✓	INS#LINK	6	D			
		✓	(machine instructions)****					
		✓	INS#TERM	0	7 F E			
	(0, name)		OPT#LINK P, REP=0	7	P			
	(name) or (1,name)		OPT#LINK P, REP=1	8	P			
	(2,name)***		OPT#LINK P, REP=2	9	P			
(name)	H,L,T,hex,char		CLS#SCAN T,X,N,H,L	T	X	N	H	L
	beg. of right part		PROLOG R, Z	Y	A			

Restrictions

* Preceded by <name> or |name| as a left part
** Must be only term in a right part and must be followed by another right part
*** Must be last term in a right part
**** Should contain no relocatable entities; base register 10 set to address of first instruction following INS#LINK

Legend

P - Production number
X - Displacement in the packed character table
N - Length of string in the packed character table
S - Semantic module number
D - Offset to beyond the BR 14 generated by INS#TERM
T - 0 or 1, depending on whether unlike or like
H - Maximum string length
L - Minimum string length
Y - Right-hand part length
A - Offset to ambiguity end
R - Current right part number
Z - Next right part number

Figure 9. Table of Correspondences in ACTION Lists

contents of the halfword loaded in a register from that location then serve as an index based on a register pointing to the ACTION list proper (WORK#TB).

2.2.2.5 CHAR#TB. The packed character table (PCT) is a string of bytes, substrings of which form comparison arguments and TRT table arguments. The PCT is entirely derived from character string symbols given as right-part terms in the BNF and from the hexstring and charstring parameters of terminal productions. the two ACTION list macros that refer to the PCT are

```
CHR#SCAN X,N
```

and

```
CLS#SCAN T,X,N,H,L
```

where X is the displacement into the PCT and N is the byte length of the PCT substring.

The PCT derives its name from the fact that attempts are made to keep substrings within substrings to save space. (For example, "USET" yields the strings "USE" and "SET.")

2.3 THE PARSER

2.3.1 General Description

This parser is a re-entrant, 1,350-byte routine written in PL360. The parser's function is to break down the input stream into its component parts (decomposition and recognition) using the ACTION list as a guide, and to delineate the input for semantic routines whenever processing is to be done. The parser can be thought of as an interpretive driver (mainline module) for the SPIRES II subprocessor.

Central to the operation of the parser is the parser table, in which is stored the status information requried by the parser; each user has his own parser table in user memory (see section 2.8). A basic component of the parser table is the production stack. This LIFO stack is a twenty-element array, with stack elements consisting of six halfwords. The current stack element is referenced using the AL (ACTION level) register. Each time a right-part term calls another production, the AL register is incremented by 1. Figure 10 is a representation of the production stack.

Before the parser is started up for a user, semantic process 0 (SEM0) is called by INIT. TIPA, SO(0), TIPB, ST(0), IL, ES(0), LAH(0), AL, SS(0), and SE(0) must be set to initial values. (See section 2.8, THE PARSER TABLE, for an explanation

	AL=1	AL=2	. . .	AL=20
Current production number				
Current right-part number				
Current right-part term number				
Input pointer at entry to production				
Relative address of currently active prolog				
Relative address of current ACTION list position				

Figure 10. Parser Production Stack

of these terms.) Also, INIT must make absolute the pointer terms of LINK#TB and store them in TV, AV, PV, and CV, respectively, of the parser table. SEMANTIC, the first word of the parser table, must contain the address of SEMANT, the collection of semantic processes (see section 2.4).

When the parser is first entered, R8 (called P) contains the production number of the first production to receive control. If a user has just logged on through INITIAL, P will contain 1. The parser begins by initializing IL2 and register zero. Register usage is as follows:

ZERO	(R0)	Contains 0.
TIP	(R1)	Temporary input pointer used for based variables.
PTR	(R2)	Pointer register for based variables.
TEMP	(R3)	General purpose.
AL	(R4)	Maintains action level. PROD, PLOG, POS, IS, PLEV, and XLEV are controlled by AL.
IL	(R5)	Maintains input level. Input level varies as parsing switches from one input medium to another. Controls SS, SE, and LAH.
IL2	(R6)	Contains IL+IL. Controls ES and SIP. SO referenced by IL2+IL2.
X	(R7)	Maintains relative location within the ACTION list for the currently active production.
P	(R8)	Contains current production number, PCT displacement, or semantic number.
FIP	(R9)	Maintains the current relative position in the input data being parsed. For any particular production, FIP's value lies within the range defined by IS(AL) through ES(IL2).

2.3.2 Changing Input Levels

It may be desirable during the parsing process to change "input channels"; that is, to switch to a new source of input. An example in the SPIRES II command language is the ADD, SUBSTITUTE, TRANSFER,...UPDATE sequence, where on recognition of the string UPD(ATE) the input pointer is switched to the WYLBUR working data set. This process is controlled by the semantics. The following demonstrates the process.

> Assume that all registers are saved on entry to the semantic module and are accessible; i.e., that SFIP is where FIP was saved, etc.

```
SEMnn:   TEMP :=IL2 + IL2;
         ES(IL2+2):=ZERO;
         SIP(IL2):=FIP;
         IS(AL):=ZERO;       SFIP:=ZERO;
         TIP:=@NEWINPUT;     TIPA:=TIP;
         SO(TEMP+4):=TIP;
         PTR:=@ST+BL;        SET(CODE(1));
         PTR:=@SS+(IL);      SET(CODE(1));
         PTR:=@SE+(IL);      RESET(CODE(1));
         PTR:=@LAH(IL);      RESET(CODE(1));
         TEMP:=BL+1;         BL:=TEMP;
         IF ZERO < PROD(AL) THEN BEGIN
            TEMP:=NEG PROD(AL); PROD(AL):=TEMP; END;
         IL:=IL+1;           IL2:=IL+IL;
         SIL:=IL+1;          SIL2:=IL2;
         GOTO EXIT; COMMENT EXIT BACK TO PARSER;

COMMEMT  CODE is defined BYTE CODE SYN MEM(PTR),
         FIP,IL, and IL2 are changed on exit.
```

2.3.3 Calling Semantic Modules

The parser calls semantic modules by loading register P with the semantic process number S contained within the ACTION list element SEM#LINK and then issuing the call. The sequence is:

```
TIP:=ZERO;
R10:=SEMANTIC;
CALLS (R14,R10); <note: this is equivalent to a BALR>
```

2.3.4 Input Bounding

When a command input has parsed successfully and all necessary processing has been performed, the command string may be "thrown away." The semantic module that is called to read a command input line will not physically read anything except unparsed input. When an input line is read, the production that calls the "read input" semantic module is marked in a certain way; the production is then said to be "bounded." If the bounded production succeeds, then the line just parsed is thrown away. If it does not succeed, the line is retained. The "read input" semantic module bounds the production that called it by negating P(AL).

2.4 SNAP

Within the parser there are calls to SNAP that can be turned on by the system programmer who issues the command

```
PATCH CORE SNAP+9 00
```

That person then receives at his terminal a term-by-term tracing of the parsing process. Figure 11 contains an example of the tracing. As with a conventional programming language, BNF grammars must be debugged and optimized. The former is of course assumed, but the latter is extremely important, especially with a "top-down" parser like ACTION. With this sort of parser it is possible to write grammars that are horrendously inefficient to parse; but it is also possible, by using some of the ACTION extensions, to write a semantically assisted grammar that parses with very little backtracking. It is necessary, however, to be able to "see" the parsing in progress, and to be able to compare command parsing that is done with two or more variants of a grammar. These requirements are met by SNAP.

2.5 SEMANT

SEMANT is a PL360 global procedure that is composed of n parts:

Declarations
Local Procedures
Group Router
Semantic Process Router, Group#1
Semantic Process Group#1
.
.
.
Semantic Process Router, Group#n
Semantic Process Group#n

Each part is discussed below.

2.5.1 Declarations

The names of external procedures called from SEMANT are declared first. These procedures are the various entry points within the ORVYL interface group, i.e., TGET, TPUT, QUIT, etc. After them, the PL360 SPIRES functions (see Appendix K) are declared. Also, the standard register names for the parser (see section 2.3.1) are declared, a dummy section for the parser table is declared based on R11, and a dummy section for the master table is declared based on R12. (These last two items are explained in more detail in sections 2.7 and 2.8.)

2.5.2 Local Procedures

These are procedures that are used by several semantic processes, such as MOVECORE (a generalized move routine); SHELSORT (an in-core sort routine); and BCSRCH (a routine to search the mnemonic dictionary). These and other local procedures are to be found in Appendix H.

```
-? display file ipf
```

① * = call
R = return

② ACTION list code

③ P, S, or X if col. ① = *. If col. ① = R, 0 = fail, ① = success

④ Production making call or being returned. If (-), production is bounded

⑤ Right-part number of col. ④

⑥ Term number of col. ⑤

⑦ Saved value of input pointer for col. ④

⑧ Input pointer current value

⑨ Maximum value that input pointer may assume; may vary as col. ⑩ varies.

⑩ Input level; varies on lookahead and redefinition.

⑪ Bounding level; varies with col. ⑩, and when productions change between bounded and unbounded.

⑫ ACTION level: current depth within production calls.

⑬ Bounding state

⑭ String start flag

⑮ String end flag

⑯ Lookahead flag

⑰ Graphic representation of ACTION level

```
 R   [illegible]    1    7-   1   1    0    0   16   0   1   2   0   1   0   0    *
 *   [illegible]  [illegible]    7-   1   2    0    0   16   0   1   2   0   1   0   0    *
 *   [illegible]  110    7-   1   3    0    3   16   0   1   2   0   1   0   0    *
 *   [illegible]  172   110   1   1    3    3   16   0   1   3   0   1   0   0     *
 R   [illegible]    1    7-   1   3    0    7   16   0   1   2   0   1   0   0    *
 *   1  134    7-   1   4    0    7   16   0   1   2   0   1   0   0    *
 R   1    1    7-   1   4    0    8   16   0   1   2   0   1   0   0    *
 *   1  [illegible]    7-   1   5    0    8   16   0   1   2   0   1   0   0    *
 *   [illegible]  [illegible]    59   1   1    0    8   16   0   1   3   0   1   0   0     *
 *   1  [illegible]    59   1   2    0   12   16   0   1   3   0   1   0   0     *
 *   5   11    96   1   1   12   12   16   0   1   4   0   1   0   0      *
 R   5    1    96   1   1   12   12   16   0   1   4   0   1   0   0      *
 *   1  134    96   1   2   12   12   16   0   1   4   0   1   0   0      *
 R   1    1    96   1   2   12   13   16   0   1   4   0   1   0   0      *
 *   1  127    96   1   3   12   13   16   0   1   4   0   1   0   0      *
 R   1    1    96   1   3   12   16   16   0   1   4   0   1   0   0      *
 *   5   11    96   1   4   12   16   16   0   1   4   0   1   0   0      *
 R   5    1    96   1   4   12   16   16   0   1   4   0   1   0   0      *
 *   [illegible]  134    96   1   5   12   16   16   0   1   4   0   1   0   0      *
 R   [illegible]    0    96   1   5   12   16   16   0   1   4   0   1   0   0      *
 R   1    1    59   1   2    8   16   16   0   1   3   0   1   0   0     *
 *   5    6    59   1   3    8   16   16   0   1   3   0   1   0   0     *
 R   5    1    59   1   3    8   16   16   0   1   3   0   1   0   0     *
 *   5   24    59   1   4    8   16   16   0   1   3   0   1   0   0     *
 R   5    1    59   1   4    8   16   16   0   1   3   0   1   0   0     *
 R   1    1    7-   1   5    0   16   16   0   1   2   0   1   0   0    *
 R   1    1     2   1   1    0    0    0   0   0   1   0   1   0   0   *
 R   7    1     1   1   1    0    0    0   0   0   0   0   1   0   0  *
 *   1    7     2   1   1    0    0    0   0   0   1   0   1   0   0   *
 *   5    1     7   1   1    0    0    0   0   0   2   0   1   0   0    *
-?
```

Figure 11. SNAP--ACTION Parser Trace

2.5.3 Group Router

It will be convenient during implementation to categorize semantic processes according to function (search, update, miscellaneous, etc.). The semantic process numbers will be arranged in such a way that the semantic processes for each function occupy a subset of contiguous process numbers. Such a subset is called a "group."

The group router is simply a series of "if" statements that route control to the appropriate process router, depending on the range that P falls into:

```
          IF P > 14 GOTO GROUP4;
          IF P > 27 GOTO GROUP3;
          IF P > 49 GOTO GROUP2;
     GROUP1:
```

2.5.4 Semantic Process Router

For any group of semantic processes, the semantic process router routes control to the process whose number is in P. This is done using a simple branch table:

```
     GROUP2:
               P:=P - 14 SHLL 2
               BRANCH (SWITCH(P));
               GOTO SEM14;
               GOTO SEM15;
                    .
                    .
                    .
```

2.5.5 Semantic Process Group

Each semantic process can either be unique or can share portions of other semantic processes within a group or a set of groups. All returns to the parser are effected by the statement "GOTO EXIT;". Figure 12 shows shared semantic processes. The vertical lines represent code between specified statement labels, and the horizontal lines represent transfers ("GOTO's").

2.6 THE SAVE STACK

(The discussion shifts in this section to a consideration of the user memory area. This and the following two sections describe the save stack, the master table, and the parser table.)

A stack of 15 register save areas that consist of 16 fullwords per area is set aside in each user memory workspace. Each time a PL360 global procedure is called, or an Assembly Language

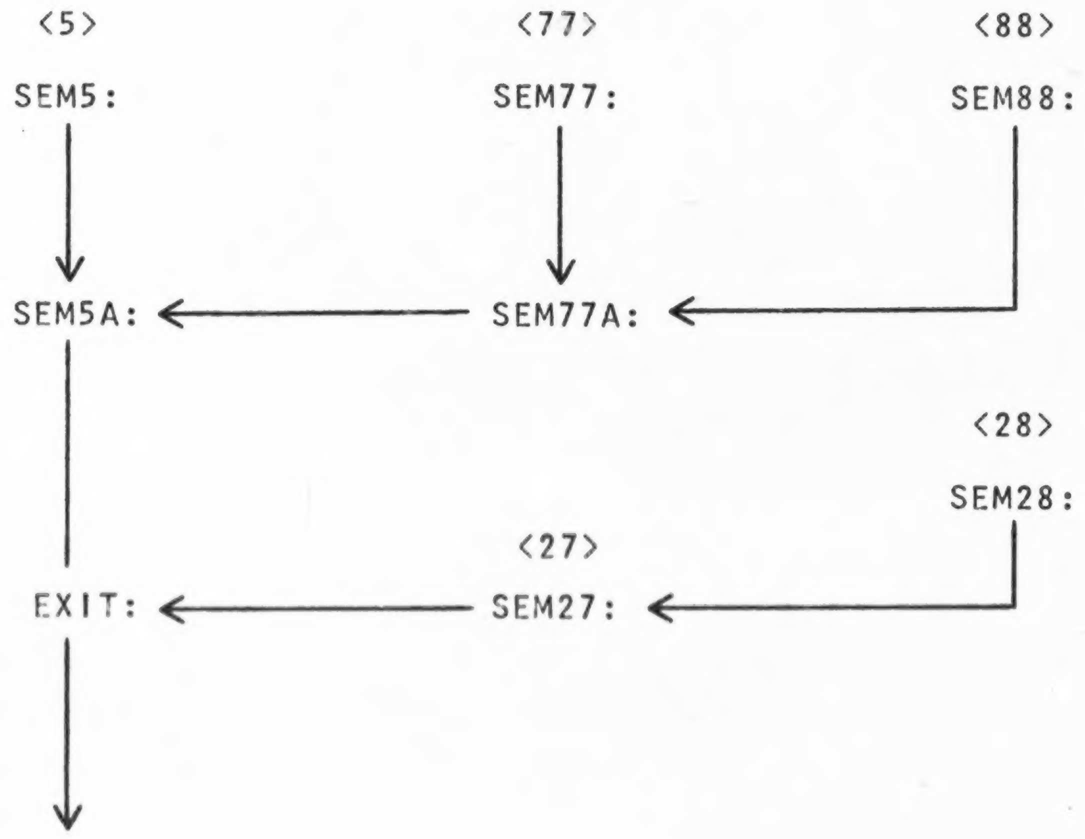

<5>
<77>
<88>
SEM5:
SEM77:
SEM88:
SEM5A:
SEM77A:
<28>
SEM28:
<27>
EXIT:
SEM27:

Figure 12: An Example of Shared Semantic Processes

entry point, the following PL360 code (or its Assembly Language equivalent) is executed:

```
          STM(R1,R0,R13);
          R13:=R13++#01000040
```

The "++" indicates a logical addition of 64 to the current value of R13, thus pointing it at the next save area in the stack. R13 is reduced by the same amount on exiting.

2.7 THE MASTER TABLE

By convention, R12 always points to the master table. This table contains user identification information, the user's terminal ID, the ID of the file currently attached, and so on.

2.7.1 TSAVER

Thirteen doublewords used by the timer and attention interrupt routines.

2.7.2 TI

A fullword containing the user's terminal ID.

2.7.3 DI

A fullword containing the user's file ID.

2.7.4 NXTSPACE

A fullword containing the address of the next space available for use in the user memory area. This cell is added to by the ORVYL interface routine "GETCORE," and reduced by the routine "RETCORE."

2.7.5 FLAGS

A fullword whose four bytes are used for various flag conditions, such as "ATTN has occurred at the terminal," or "time-out has occurred."

2.7.6 USACCT

A fullword containing the account number given by the user when he logged on.

2.7.7 USPSWD

A fullword containing the terminal identification name (e.g. P24) that corresponds to the terminal at which the user is logged on.

2.7.8 960-Byte Reserved Area

An area reserved for the growth of the master table. It is probable that a large portion will be devoted to files devices work space and to counters in which on-line statistics will be accumulated.

2.8 THE PARSER TABLE

This table contains the constants and variables required by the parser during its execution on behalf of a particular user. The following is a list of table elements and their lengths and descriptions.

PARSER TABLE: VARIABLE NAMES + FORMAT (R11-BASED)

VARIABLE NAME	SIZE	DESCRIPTION
1. SEMANTIC	(4)	Absolute address of the semantic module associated with the ACTION list being parsed.
2. TV	(4)	Absolute address of the terminal productions in the ACTION list.
3. AV	(4)	Absolute address of the nonterminal productions in the ACTION list.
4. PV	(4)	Absolute address of the production table.
5. CV	(4)	Absolute address of the character table.

<The preceding five terms must be pre-set before calling the parser. The method of establishing TV, AV, PV, and CV is described in section 2.9.2. The remaining terms of the parser table may be in any order, but are currently defined as follows:>

6. TIPA	(4)	Absolute address of the input data being parsed (upper case only).
7. TIPB	(4)	Absolute address of upper/lower-case input data. Not used by the parser.

8. CHARSTART	(4)	Pointer to the beginning of the character string most recently parsed by a terminal production.
10. CHARPOS	(2)	Number of bytes last parsed by a terminal production.
11. PLOGX	(2)	Relative address of the current production prolog.
11. BL	(2)	Bounding level counter--initialized to 0 by parser.
12. WS2	(2)	Work space for parser.
13. LNTH	(2)	Used by terminal production process to hold size information.
14. MAXC	(2)	Used by terminal production process to hold H.
15. MINC	(2)	Used by terminal production process to hold L.
16. CONF	(2)	Used by terminal production process to hold function code.
17. PROD	(20*2)	Array that maintains production numbers currently in process. If a number in this array is negative, the corresponding production is bounded, and BL was incremented when bounding occurred.
18. PLOG	(20*2)	Array that maintains relative address of prologs currently active.
19. POS	(20*2)	Array that maintains relative address in the ACTION list of productions in process.
20. IS	(20*2)	Array that saves relative input pointer at entry to each production.
21. PLEV	(20*2)	Array that maintains right-part number within a production (may be eliminated).

22. XLEV	(20*2)	Array that maintains term number within a right part (may be eliminated).
23. SO	(4*4)	Array that maintains TIPA as input levels vary.
24. ES	(4*2)	Array that maintains length of input data being parsed at the various input levels.
25. SIP	(4*2)	Array that maintains relative input pointer as input levels vary.
26. SS	(4*1)	Array of flags that specifies if start of input data is static (00) or flexible (FF).
27. SE	(4*1)	Array of flags that specifies if end of input data is static (00) or flexible (FF).
28. LAH	(4*1)	Array of flags that specifies if a lookahead is in progress (FF) or not (00).
29. ST	(8*1)	Array of flags that maintains bounding state: (00) = normal bounding; (FF) = bounding input level change.
30. CAOF	(256*1)	Array of (00) used by terminal production process.
31. CAON	(256*1)	Array of (FF) used by terminal production process.
32. CHAR1	(1)	Flag byte used by terminal production process.
33. CHAR2	(1)	Flag byte used by terminal production process.
34. RS	(1)	Return Switch flag. (00) = production successful; (FF) = production failed.

2.9 THE FLOW OF CONTROL

The following sections refer back to Figure 5. The last number of each section corresponds to a numbered arrow in Figure 5.

2.9.1 Entry to SPIRES II

The user has logged on to the system through MILTEN, and issued the command "SPIRES."

2.9.2 INITIAL

INITIAL then initializes the master table, finds the address of the first page of user memory, and initializes for communication with WYLBUR. It also initializes the parsing table by adding the ACTION list origin address to the LINK#TB vector to get TV, AV, PV, and CV.

2.9.3 Call to SEMO

SEMO is a semantic process that initializes the parser. After doing so, it returns to INITIAL.

2.9.4 Call to the Parser

The P register is set to 1 and the parser is called. The first right part of production 2 has a call to SEM1, which, by convention, reads a line of input. The parser remains in primary control until a LOGOFF command is recognized.

2.9.5 Calls to Semantic Processes

Each time the parser finds a semantic link call, it loads the P register with the production number from the ACTION list and calls SEMANT.

2.9.6 Calls to the ORVYL Interface

Semantic modules issue calls to ORVYL interface entry points whenever supervisor services, such as disk or terminal input or output, or WYLBUR communication, are required.

2.9.7 QUIT

The parser has recognized LOGOFF and therefore does a return (BR 14); or, an error has occurred and the parser gives the user back to WYLBUR by calling QUIT.

2.9.8 Branch to SNAP

Whenever the branch-condition-15 instruction at SNAP + 9 has been altered to branch condition 0, a trace line is printed by SNAP each time a production is called or a return is made to a calling production.

CHAPTER 3

LOGICAL FILE CONCEPTS

3.1 INTRODUCTION

This chapter and the three following describe the SPIRES II file structure. This chapter introduces the concepts and terminology required for a basic understanding of the general structure. It does not attempt to outline the structure in all its physical detail, but rather attempts a logical presentation. The concepts "record," "record type," "profile," and "record characteristics" are emphasized.

Chapter 4 explains how data are organized to facilitate economic storage and easy access. A wide range of storage strategies has been provided to accomodate both simple and sophisticated applications.

Chapter 5 deals with the physical formats of the entire data structure: records, blocks, characteristics tables, and so forth. This chapter is not required reading for an interested observer of the system, but it is vital for anyone wishing to write programs, modules, or subroutines that manipulate SPIRES II files.

Chapter 6 presents the structure of the Basic Files Services software. This software, used in conjunction with ORVYL macros (see Appendix D) or the virtual access method (see section 1.1.5.3) constitutes an access method for locating, adding, replacing, or deleting records within a SPIRES II file.

3.2 FILE SYSTEM DESIGN REQUIREMENTS

In order to provide SPIRES II with the functional capabilities deemed necessary by its designers, a file structure capable of handling a wide variety of applications was conceived. Following are some of the major requirements for such a file structure.

> It must be possible to search the file through a number of access paths. Where files are too large for sequential searching, it must be possible to build quick access paths using any data elements desired. Desired data should be locatable through such paths in an average of three to four disk accesses, depending on file size.
>
> The file must be able to accomodate data elements of indeterminate length with reasonably efficient use of space.

The file must be able to accomodate optional and multiple occurrences of data elements within records.

The file owner must be able to define the characteristics and contents of his file, within practical limits, without the intervention of programming personnel.

It must be possible to modify a record at the most basic level, i.e. by character.

The file structure must interface easily with special data element transformation algorithms and tables.

The structure must be able to encompass large, growing files. Documented requirements include a catalog data file, which will grow at the rate of 50,000 records per year to an ultimate size of 250,000 records.

It must be possible to provide file security at the data element level.

The system must attempt to provide absolute integrity among data entering the system. No user data should be permanently lost by the system.

The structure must allow for a variety of recovery techniques, ranging from the simple overwriting of one byte in a file to the complete reconstitution of the file from previously gathered information.

Response time, which is tied to the number of disk accesses required to service a request, must be minimized.

There must be sufficient areas of redundancy built in to permit programmatic file verification and reconstruction of certain parts of user files.

Time intervals during which a file's integrity is unprotected must be eliminated or significantly reduced.

To reduce the number of accessing routines necessary, data structures must be as few and as general as possible.

There must be options for holding frequently used tables and dictionaries in user virtual memory in order to reduce accesses to disk.

3.3 FILE STRUCTURE OVERVIEW

To someone familiar with classic computer file organizations that feature "indexes, "prime data records," "inverted lists," and so on, the terminology used to describe the SPIRES II file structure may seem strange. Nowhere is the word "index" mentioned; nowhere is the phrase "prime data record" used. The reason for these omissions is simple. Such terminology suggests the narrower consideration of a two-level file structure: a series of prime data records pointed to by one or more indexes of data element values. SPIRES II, although it is designed to efficiently accomodate one level of indexing, is not limited to this. (See Figure 13.) The concepts that are presented here relate to files composed of n-level hierarchies of record types, and to "profiles" that define the "window" on a file open to a particular set of users. These profiles not only define the levels of the hierarchy to which the set of users is allowed access, but also the portions of each record type that the users are allowed to access or modify. Also presented in this chapter are brief introductions to the concepts of data element structure within a record type and file characteristics that enable the generalized system software to operate on different files, each having a different structure and format.

3.3.1 Record Types

A file is defined as a collection of one or more record types, stored with characteristics information that describes those record types, how they may be searched, and how data are passed between them. All the records of one type are stored together in a single data set. Within the system, each record type is assigned a number from an integer code 1, 2, ...n, and is referred to as REC1, REC2, etc.

3.3.2 Pointers

Records of one type may refer to records of another type by means of pointers (see Figure 14). Records of one type may not point to records of more than one other type, but, as shown in Figure 14, one type of record may be pointed to by more than one other type. In one method of record organization, it is possible to form hierarchies of record types that point to one another, with no theoretical limit to the number of levels possible.

3.3.3 Goal Records

If a certain type of record is to be retrieved, the records of that type are called "goal records." A record that is pointed to by records of another type is also called a goal record. Goal records are usually the basis for creating access records.

3.3.4 Access Records

Types of records that point to goal records are called "access records." However, an access record may also contain data

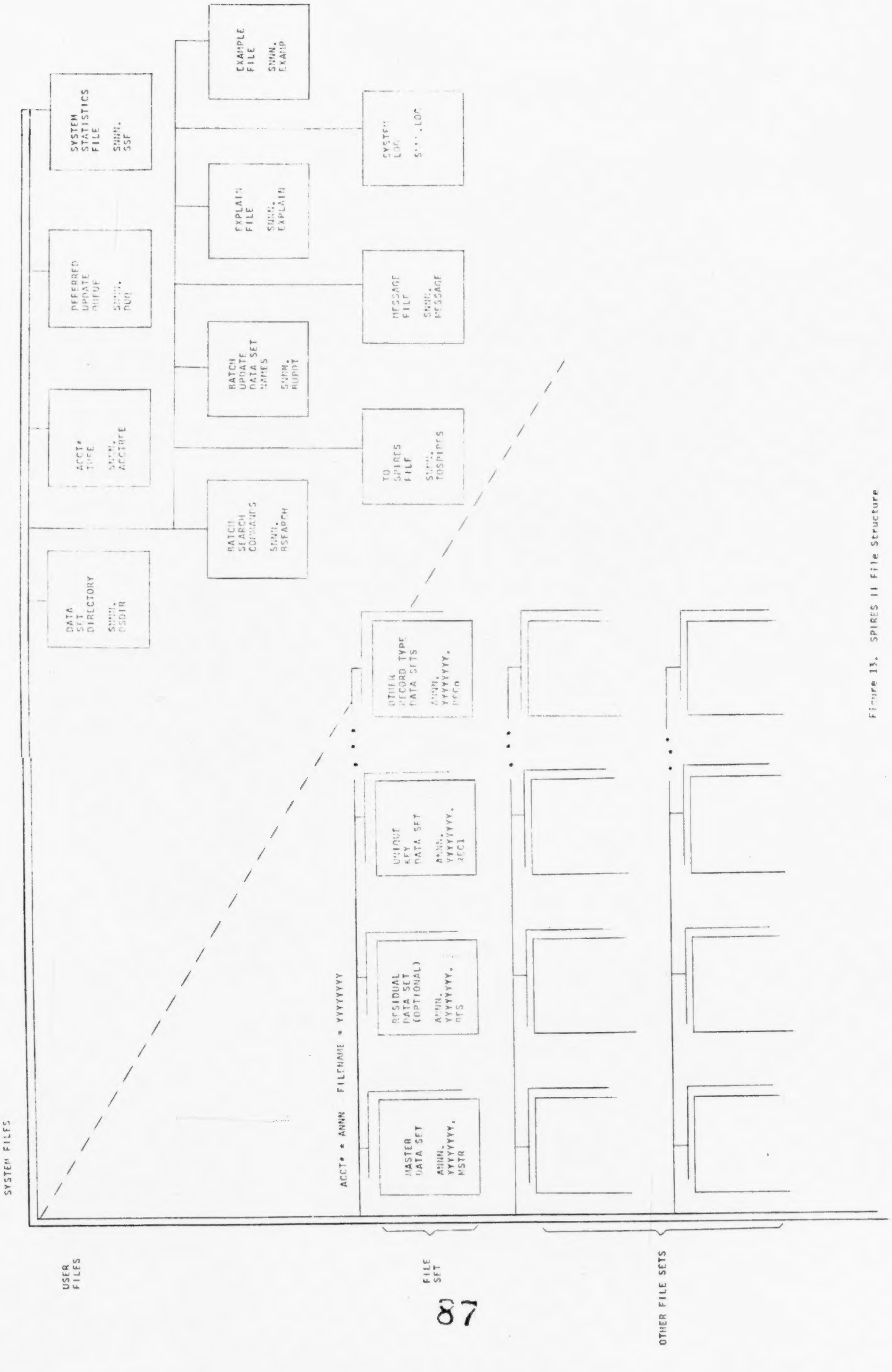

SYSTEM FILES
USER FILES
DATA SET DIRECTORY SNNN. DSDIR
ACCT# TYPE SNNN. ACCTYPE
DEFERRED UPDATE QUEUE SNNN. DUQ
SYSTEM STATISTICS FILE SNNN. SSF
BATCH SEARCH COMMANDS SNNN. BSEARCH
BATCH UPDATE DATA SET NAMES SNNN. BUPDT
EXPLAIN FILE SNNN. EXPLAIN
EXAMPLE FILE SNNN. EXAMP
TO SPIRES FILE SNNN. TOSPIRES
MESSAGE FILE SNNN. MESSAGE
SYSTEM LOG SNNN.LOG
ACCT# = ANNN FILENAME = YYYYYYYY
MASTER DATA SET ANNN. YYYYYYYY. MSTR
RESIDUAL DATA SET (OPTIONAL) ANNN. YYYYYYYY. RES
UNIQUE KEY DATA SET ANNN. YYYYYYYY. REC1
OTHER RECORD TYPE DATA SETS ANNN. YYYYYYYY. RECn
FILE SET
OTHER FILE SETS

Figure 13. SPIRES II File Structure

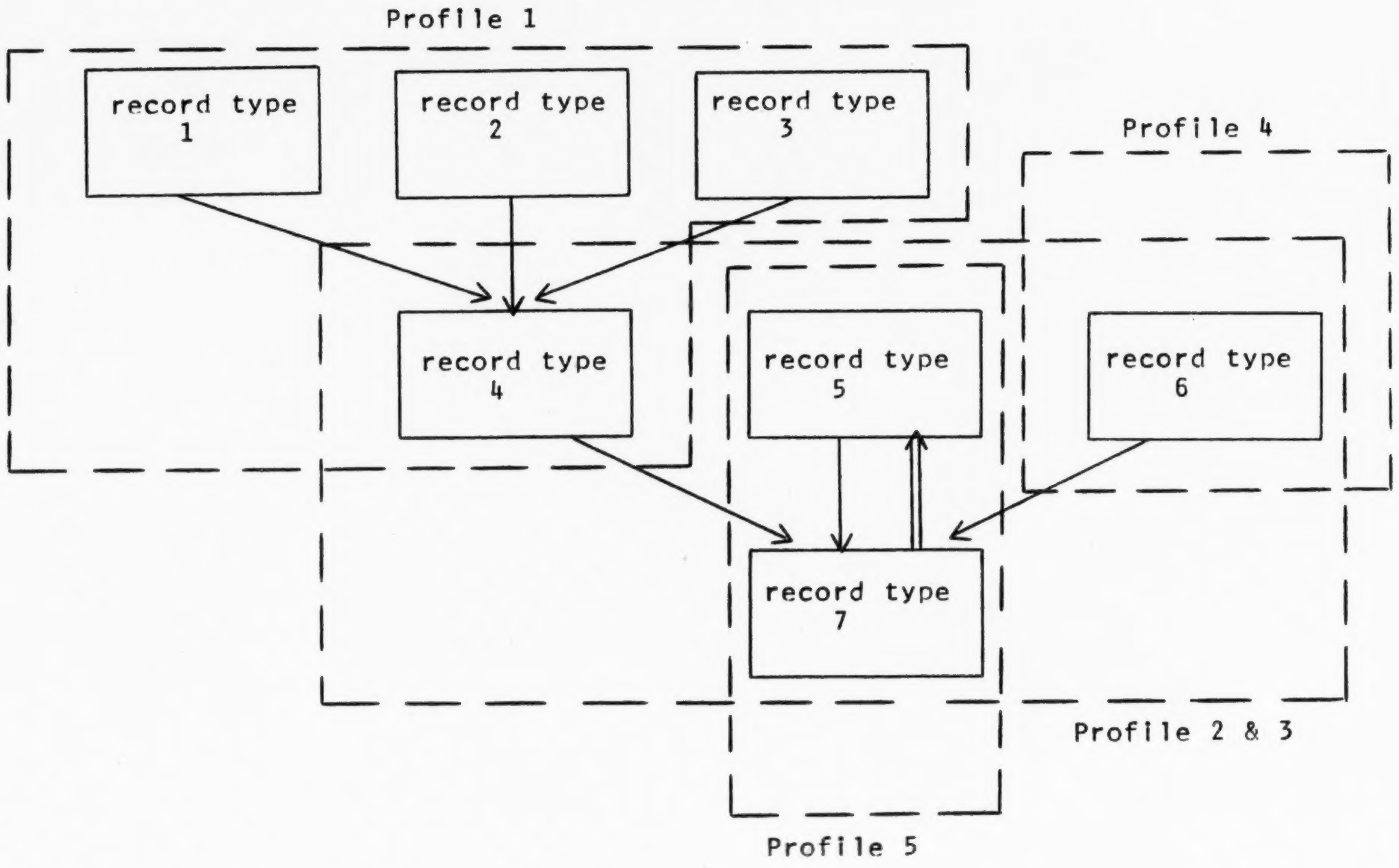

Profile 1
record type 1
record type 2
record type 3
Profile 4
record type 4
record type 5
record type 6
record type 7
Profile 2 & 3
Profile 5

Figure 14. Relationships Between Record Types

to be retrieved; in such cases the access record is also a goal record.

3.3.5 Record Contents

Within a record of a particular type, the following components may exist.

> THE KEY DATA ELEMENT. Each record of a specific type must contain a key data element whose value is unique among the records of that type. This key data element value distinguishes one record from another and it is the basis for organizing record data sets for quick access (see section 4.4). It is common practice to refer to a particular record type by the name of its key data element--e.g., the Author record type.
>
> THE POINTER GROUP. If a record data set consists of access records, then the records in that data set may contain pointer groups. An access record pointer group is a set of pointers to the goal records that contain, or have some association with, the key data element values of the access record. (These pointer groups are furnished by the goal records--see section 3.3.6.) Each pointer may be accompanied by qualifying information that identifies the goal record pointed to as a member of some class, range, or set. Pointers may consist of block number (see section 4.2.3) or node or entry number references (see section 5.2) or of key data element values. Pointer groups are maintained in ascending order according to the collating sequence.
>
> DATA. This important component consists of all the values of data elements that are associated with a particular key data element value. A record does not always contain data; it could be absent, for example, from a record type like number 1 in Figure 14. Records of this type only refer to records of another type; they are not themselves the goal records of any other record type.

Of these three components, the key data element must always be present. The pointer group must exist if the record type is an access record and refers to a goal record outside itself. The data portion should exist in all goal records, and may exist in access records as well.

3.3.6 Passing Data Element Values

Key data element values and pointer groups are added to access records by means of "passing." Each goal record passes over to the appropriate access record a data element value that will be the key data element in the access record. Also passed is a pointer back to the goal record and any data element values to be used as qualifiers.

The algorithm for passing from goal record to access record is as follows. If among the access records to receive the "passed" data element value there is none that has as its key data element value the value that is being passed by the goal record, a new access record is created. This new access record includes a single pointer back to the goal record (together with any qualifying information, this will be the pointer group). If an access record already exists with the key data element required, then the pointer passed upward from the goal record (along with qualifying information) is merged into the pointer group in the existing access record; the collating sequence within the pointer group is maintained throughout.

3.4 PROFILES

When a user wishes to search or update a file, he is dealing with a goal record type and zero or more access record types. The requirements of one or more users with respect to certain sections of a file are called a profile. This word is more clearly defined to mean a subset of the record types (a goal record type and one or more access record types); a set of restrictions on searching and updating records in this subset; and a set of account numbers of the users that are allowed to use the file in the way the profile permits. Figure 14 shows a hierarchy of record types. In profile 1, record type 4 is the goal record and record types 1, 2, and 3 are the access records. In profiles 2 and 3, record type 7 is the goal record and record types 4, 5, and 6 are the access records. The difference between profile 2 and profile 3 lies in the update and search privileges granted to the two corresponding user groups. Profile 4 is interesting in that there record type 6 is both a goal and an access record. Profile 5 is even more interesting: record type 7 is the access record, and record type 5 is the goal record. This is a reversal of the relationship between these two records in profiles 2 and 3. The double arrow indicates the second set of pointers. An implicit limitation in defining profiles is that no more than one goal record and two levels can be included within one profile.

A user must select the profile in which he wishes to work by issuing a SELECT <name> command. Until he overrides this with a LOGOFF or another SELECT command, he may only operate under the record type subset and accessing restrictions dictated by his profile. If he should select a profile that does not exist under his account number, he receives an error diagnostic.

3.5 THE HIERARCHICAL STRUCTURING OF DATA ELEMENTS

Up to this point, hierarchies of record types have been the only ones discussed. It is also possible, however, to have hierarchies of data elements within a record type.

Each record type may be considered the top level of an n-level data element hierarchy. In a hierarchy of only one level, none of the elements at that (record) level break down into subelements. In a hierarchy of several levels, there are data elements at the record level that can be broken down into other data elements. (For example, "date" may resolve into "month," "day," and "year," although structures are usually more complex than this.) Such data elements are called "structures." Data elements in a structure may themselves be structures.

Data element structures are treated as records within records. Figure 15 shows at the larger record level the record types "Author," "Title," and "Date." The Author structure is made up of "Name" and "Affiliation," and the Date structure of "Publication (Date)" and "(Date of) Receipt." The latter two are themselves composed of "Month" and "Year."

Within a record type, each structure is assigned a unique internal structure number. Each data element is referred to internally by its "structure element number," which uniquely identifies the data element and the internal structure of which it is a part.

Although the implementation limit on the depth of structuring is ten levels, it is felt that a depth exceeding three levels places a processing burden on the system that should be avoided if possible.

3.6 FILE CHARACTERISTICS

For each record type there are three tables that together describe its format, content, searchability, and updateability. The characteristics tables are stored in the master data set as the search, build, and record format characteristics. It is impossible to access or manipulate records without using these characteristics tables.

The search characteristics table lists all the access records and any other means (such as sequential scanning using something other than the key data element) for locating goal records. These characteristics also describe qualifying information

```
Record Level    Structure Level 1    Structure Level 2

AUTHOR;
                NAME =  JONES,       JOHN R.;
                AFFILIATION =        STANFORD UNIVERSITY;

AUTHOR;
                NAME =  SMITH,       ROBERT L.;
                AFFILIATION =        RAND CORPORATION;

TITLE =  REACTOR-POWERED DESALINIZATION;

DATES;
                PUBLICATION;
                                     MONTH = JUNE;
                                     YEAR = 1970;

                RECEIPT;
                                     MONTH = OCTOBER;
                                     YEAR = 1970;
```

Figure 15. Data Element Structures In External Format

contained in access records that may restrict the goal records located. If search characteristics do not exist for a record type, the records can be located only by their key data element.

The build characteristics table contains the information needed to construct the record type from data input in external format.

The record characteristics table contains information for each data element in the record type, including its location in the record, its length (if fixed), occurrence, its type, etc.

CHAPTER 4

ORGANIZATION OF DATA SETS FOR ACCESS AND STORAGE

4.1 INTRODUCTION

The preceding chapter discussed logical file entities: record types, profiles, data structures, and record characteristics. This chapter begins by describing the data management features provided by ORVYL. It then touches briefly on the various data sets that make up a SPIRES II file and the data set naming conventions. It describes how record type data sets are organized for access, and how that organization is maintained. Finally, it give an example of a typical arrangement in a bibliographic file.

4.2 THE ORVYL ENVIRONMENT

4.2.1 Data Management Under ORVYL

ORVYL, the Stanford Time-Sharing Monitor, provides basic read and write capabilities to 2314 disk in a paging environment. Logical data sets are composed of a collection of 2,048-byte blocks. Such data sets may be organized either so that their blocks are contiguous, or so that the blocks are noncontiguous. In the noncontiguous arrangement, allocations of new blocks to a user's file are made from a common pool; blocks for one logical file are stored noncontiguously, and are perhaps spread across several volumes. Although this is the most efficient use of space, it creates the need for additional access paths to relate a logical record number to a block number. For large data sets, this arrangement adds significantly to the number of accesses required to retrieve a record. It also complicates recovery sequences. For these reasons, file contents will be organized contiguously for SPIRES II.

4.2.2 Contiguous Data Sets

Data sets with contiguous blocks (hereafter called contiguous data sets) are declared using the CONTROL 21 and CONTROL 22 macros (see Appendix C). Up to 15 secondary contiguous extents (not necessarily contiguous with previous extents) may be allocated after the primary extent has been allocated. Initially, SPIRES volumes will be allocated completely as one large common data set to prevent other ORVYL users from gaining space for noncontiguous data sets. All allocations for SPIRES files will take place under ORVYL during second or third shift, at a time when no users are on-line, and no batch activity is in progress. The sequence of events will be

(1) deallocate a group of blocks from the "pool" data set; (2) allocate that group of blocks to the new data set primary extent or the old data set secondary extent.

It should be noted that whereas ORVYL noncontiguous data sets may consist of logical records that are greater or less than 2,048 bytes, contiguous data sets must consist of logical records that are precisely 2,048 bytes. The term "logical" is used here with reference to ORVYL. SPIRES/BALLOTS will subdivide ORVYL blocks into its own logical records.

4.2.3 Accessing ORVYL Files

Accessing contiguous data sets within ORVYL is done via the standard ORVYL macros ATTACH, READ, WRITE, etc. A directory of contiguous data sets will be maintained on disk by ORVYL and kept up-to-date via the allocation mechanism mentioned above. The directory will contain information about extent limits and whether a particular file is divided into extents. The fact that files may have several extents is not apparent to the programmer; it is necessary only to provide a block number, and ORVYL will convert it to the appropriate physical record number by subtracting the number of records in previous extents and adding the result to the block number that begins the extent.

4.3 FILE SETS AND DATA SETS

4.3.1 File Sets

What the user thinks of as a "file" is, in reality, several data sets (refer to Figure 13). The file is composed of the following data sets.

A master data set that contains the characteristics of the file.

One data set for each record type in the file.

An optional residual data set, to be used in cases where it is considered economic to remove or split data away from accessing information (see sections 4.5 and 4.6).

4.3.2 Data Set Naming Conventions

System data sets are individually named on the basis of the system account number and a mnemonic that suggests the contents. These mnemonics will be discussed in further detail below.

User data sets within a file set are named with the user account number (ANNN) and the file name (up to eight alphanumeric characters) in the format

ANNN.FILENAME.DSNAME,

where DSNAME will be one of the following:

- MSTR - the master data set.
- RES - the residual data set if one exists.
- REC1 - the data set containing all records of type number 1.
- REC2...n - the data sets containing the other record types in the file.

4.4 THE ORGANIZATION OF RECn DATA SETS

The records in the data sets ANNN.FILENAME.RECN are organized either as a series of simple, fixed-length slots or in a tree structure. In either case, the organization is based on the unique key data element value in each record. Since all physical data blocks under ORVYL are 2,048-bytes long, each physical block will probably contain a number of records.

4.4.1 Slot-Structured Data Sets

Figure 16 is a graphic representation of a slot structure. The slot structure may only be used for fixed-length consecutive integer key data element values. If the records are not removed to the residual data set, the records must be fixed length. generated by the system. The primary advantage to this structure is the quick access it permits to any record. Given a key data element value and the number of records that will fit into a 2,048-byte block, one can compute the block number of the physical record that contains the desired value.

4.4.2 Tree-Structured Data Sets

When records are organized in a tree structure, they are called "nodes." Nodes are accessed by their key data element values. The "tree" refers to the logical pattern of the blocks making up the data set. One enters at a common block (the trunk) and the nodes in this block point to other blocks (branches). (The tree is in effect uprooted, being represented upside-down.) The nodes are so arranged in the tree that an examination of a trunk block determines which branch block (if any) needs to be examined next.

Consider the thumb index on the edge of a dictionary: it contains about twenty nodes that determine on which pages to begin a more discriminating search. Then the words at the tops of the pages can be used as nodes in order to determine which page to examine in detail. This example illustrates two rules

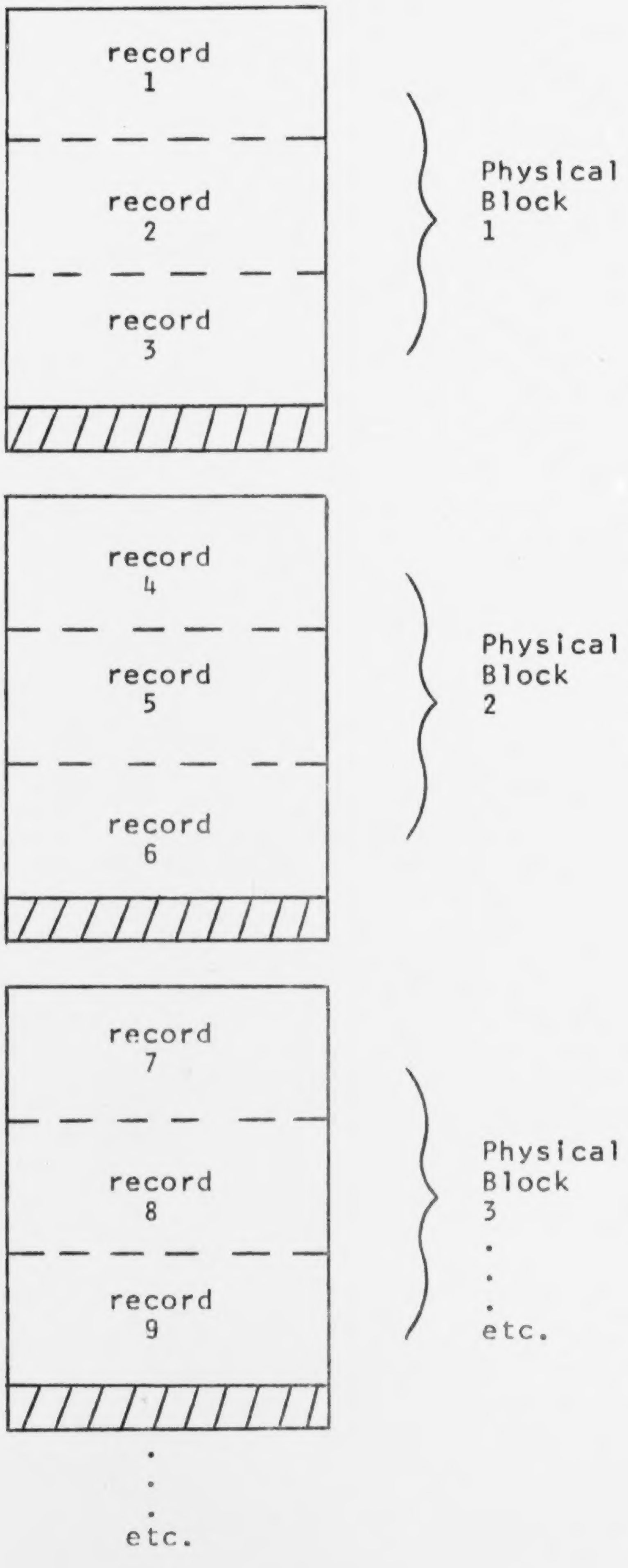

record 1
record 2
record 3
Physical Block 1
record 4
record 5
record 6
Physical Block 2
record 7
record 8
record 9
Physical Block 3
. . . etc.
. . . etc.

Figure 16. Slot-Structured Data Set

governing tree structures: (1) at any level of the tree the nodes in a branch alphabetically surround terms contained in branches emanating from that branch, or trunk; (2) a branch contains only terms that fall between alphabetically adjacent terms in its trunk.

But the tree structure used in SPIRES II data sets differs from that of the dictionary in a number of ways. First, the physical layout of a dictionary is quite different from the physical layout of the SPIRES blocks. Second, the thumb index and running head words are incomplete dictionary entries and have to be repeated on the pages; each of the SPIRES nodes is complete and so occurs only once in a structure. Third, in the dictionary example there are a fixed number of levels; in the SPIRES tree sructures there may be any number of levels, and some branches may contain more levels than others (their branches may have branches). The SPIRES tree structure grows when a terminal branch has become filled and has to start pointing to additional branches. Since growth may be uneven, it is likely that the tree will occasionally need rebalancing. This means that the tree will be reconstructed so that each branch in the tree contains about the same number of nodes. Figure 17 shows a simplified tree. A line of words makes up a block. The words are nodes and the arrows are branch pointers. Each word preceded by an extended arrow is "nonterminal"; i.e., it points to another branch (block).

4.4.3 Tree Rebalancing

The processes of rebalancing and constructing record data set trees are similar. If an old tree is being rebalanced, the old nodes are written out on tape in alphabetical order. If a new tree is being constructed, the nodes to be placed in it are sorted alphabetically. If the initial nodes are carefully chosen so that they form a balanced subset of what will probably be the ultimate node set, the tree is more likely to grow in a balanced manner. During construction or rebalancing, the nodes are placed in the tree starting at the tips of the branches (bottom-level) and working toward the trunk (top-level). Figure 18 shows the relative positions of nodes in a sample tree. For the sake of simplicity it is assumed that four nodes fit in a branch (i.e., four records in a block). Unless the tree is likely to grow heavily near the end of the sort order, the tree can be improved by bringing nodes close to the trunk. This minimizes the accesses needed to find a node. Nodes are brought toward an unfilled top level in the following ways: (1) if the farthest-left branch in the level just below the top contains nonterminal nodes, these nodes and their branch pointers are extracted, starting at the end of that block and working toward the beginning, until the higher level of the tree has been filled; or (2) if the farthest-right branch contains terminal nodes, then these nodes are extracted starting at end of that block and working toward the beginning, until either the higher level has

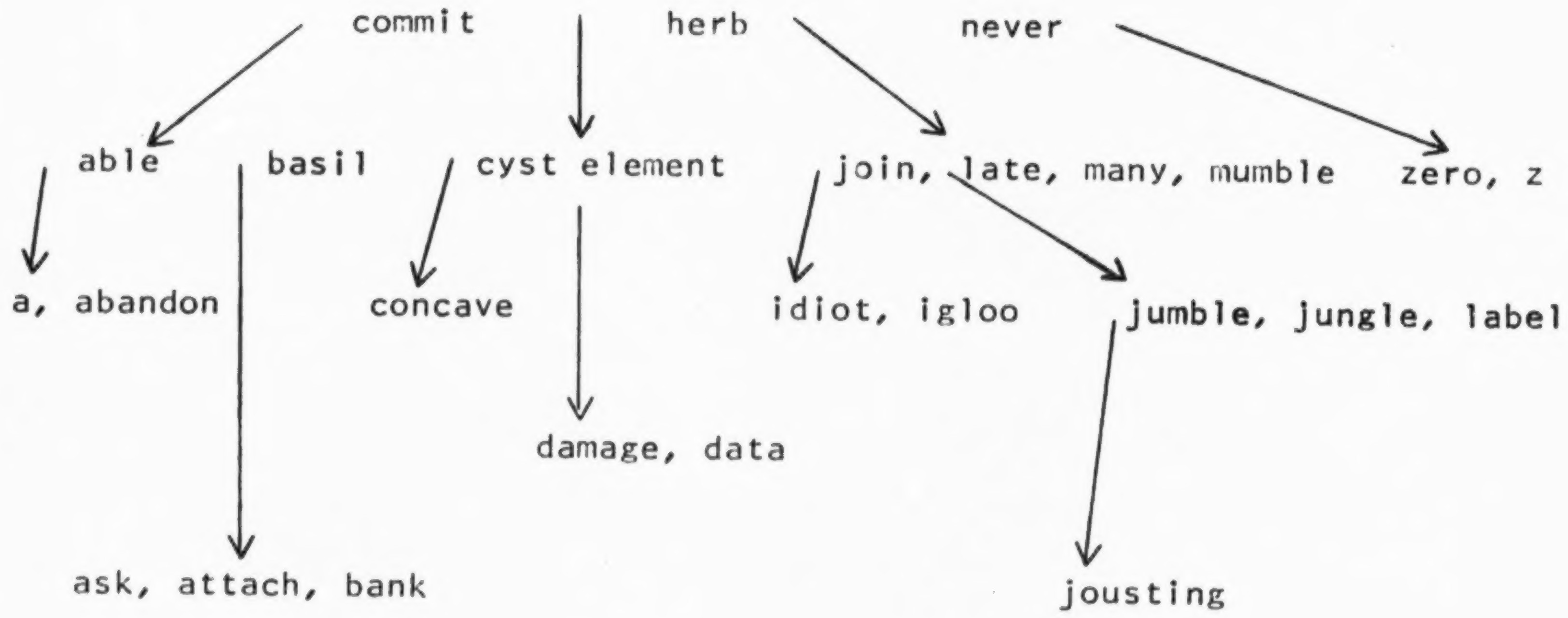

commit
herb
never
able
basil
cyst element
join, late, many, mumble
zero, z
a, abandon
concave
idiot, igloo
jumble, jungle, label
damage, data
ask, attach, bank
jousting

Figure 17. An Example of a Tree-Structured Data Set

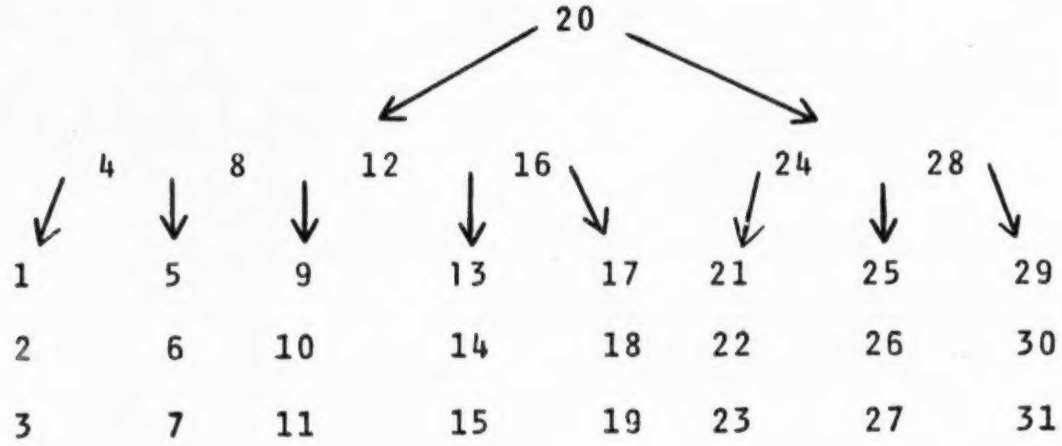

20
4
8
12
16
24
28
1
2
3
5
6
7
9
10
11
13
14
15
17
18
19
21
22
23
25
26
27
29
30
31

Figure 18. Sample Tree

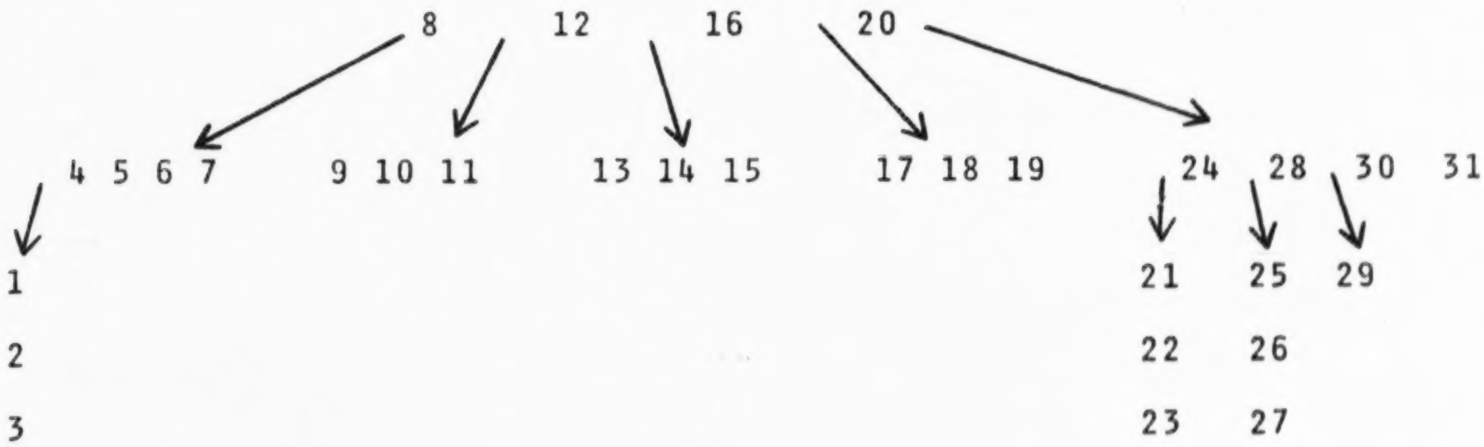

8
12
16
20
4 5 6 7
9 10 11
13 14 15
17 18 19
24
28
30
31
1
2
3
21
22
23
25
26
27
29

Figure 19. Sample Tree After Rebalancing

been filled or the lower level block has been emptied. This process fills nearly all the top level blocks. Figure 19 shows the final position of the nodes of Figure 18.

Trees are rebalanced infrequently. Between rebalancings, nodes may be deleted, modified, or inserted. Deleting is the simplest function, since all that is done is to set a status switch indicating that the node has been deleted. The actual node itself will not be removed from the tree until the next rebalancing. This allows the value of the key data element to be used for branching but precludes actual retrieval of the node. If the updating of a node involves changing the value of a key data element, then the node is set to "deleted" and an updated version of the node is inserted into the tree. If the key data element value is not changed and the updated node will occupy less space than before, then the old node is written over. (If the updated node would occupy more space, then the old node is written over with a portion of the updated node and the remainder is stored in the file's residual data set--see Figure 13.)

When a node is to be inserted in the tree, the tree is examined until a block is encountered where there is no branch pointer between the two nodes alphabetically bounding the new node. Then an attempt is made to insert the node into this block. It will sometimes happen that there is not enough reserved space left in a block for the new node. If this happens, then the contents of the block must be divided and a new block must be added to the tree. If all of the terminal nodes (including the new one) will fit into one block leaving growth space, then they are put into the new block with a branch pointer left in the old block. If they do not all fit into one terminal block, then they must be split between the old and new blocks. It is at this point that a lopsided growth factor (a file characteristics parameter entered when the file is defined) is referred to in dividing the terminal nodes between blocks. If the factor is .5, half the nodes are assigned to each block. If the factor is close to 1 (new key values are monotonically increasing), almost all the old nodes are put in the new block with little space reserved for growth. (The supposition here is that the tree will be growing horizontally and not vertically; hence there is not a requirement to provide growth space.) At this point, the first terminal node remaining in the old block has become nonterminal, since it points to the new block. If the old block has no other nonterminal nodes and if there is space in the block above this one in the tree, then the new nonterminal node is passed up to it for insertion. Otherwise it remains in the old block.

Although the algorithm as presented may appear complicated, it is relatively easy to program and has the advantage of ensuring balanced local growth. Figure 20 shows the tree presented in Figure 19 after a period of intense growth in one area. The letters were added in alphabetical order and all fall

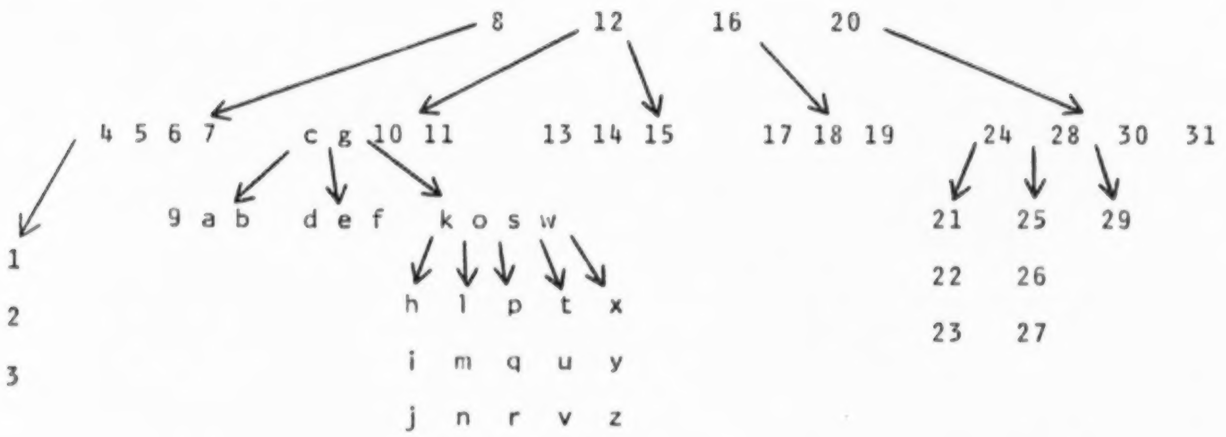

8 12 16 20
4 5 6 7
c g 10 11
13 14 15
17 18 19
24 28 30 31
1
2
3
9 a b
d e f
k o s w
h i j
l m n
p q r
t u v
x y z
21 22 23
25 26 27
29

Figure 20. Sample Tree After Intense Local Growth

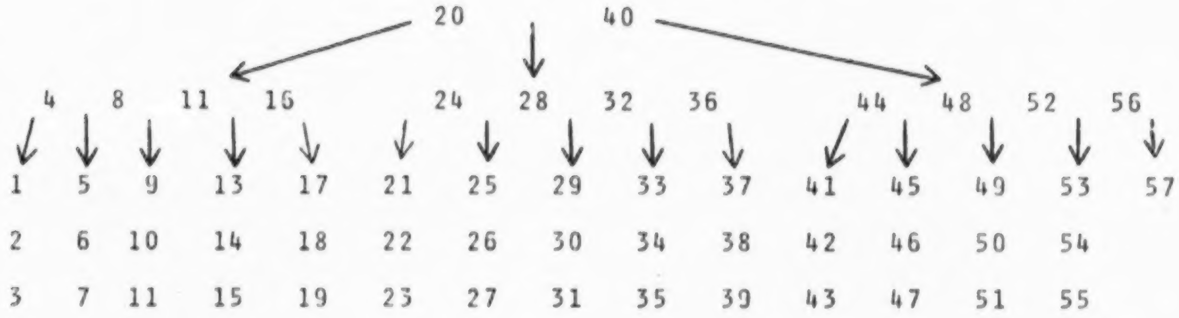

20 40
4 8 11 16
24 28 32 36
44 48 52 56
1 2 3
5 6 7
9 10 11
13 14 15
17 18 19
21 22 23
25 26 27
29 30 31
33 34 35
37 38 39
41 42 43
45 46 47
49 50 51
53 54 55
57

Figure 21. Sample Tree with Well-distributed Growth

between 9 and 10. It is anticipated that most trees, if constructed properly, will grow equally in all areas. If this is so, then trees will rarely need rebalancing.

If we had assumed Figure 18 to be a monotonically growing tree, and we had added 32-57 onto that tree, the resulting tree would have looked like the one in Figure 21. Figure 19, however, would not then represent the tree after rebalancing, since monotonically growing trees are not rebalanced in the manner shown here.

4.5 DATA REMOVAL

4.5.1 Definition and Criteria

Data removal means moving data that had been stored with an associated key data element value to another place of storage, and leaving behind a pointer to the data's new location. The residual data set ACCTN.FILENAME.RES exists to receive all such data removed from the record type data sets.

The reasons for removal are due to the fact that in a tree-structured data set (which the majority of data sets are), efficient access is produced by maximizing the number of nodes per block. In certain cases, it is most efficient to keep key data element values in a tree structure and to remove the relatively larger body of data, to be retrieved in one separate access after the correct key data element value and residual pointer have been located.

The decision algorithm for removal is based on the following:

There are N occurrences of the record in the record type data set; in order to locate a particular occurrence, some number of key data elements needs to be examined. K is the length of the key data element as a percentage of a typical record. On the average M records fit into a block; the blocks form a balanced tree structure; and it is equally likely that any record will be requested. That there are two storage strategies leads to different formulas for the expected number of accesses needed to find a record. Where the record is stored as a whole, there is a probability of M/N that only one access will be needed, a probability of (M + 1)M/N that exactly two accesses will be needed, and so on. Where only the portion needed for access is stored, with a pointer to the rest, there is a probability of (M/K)/N that only two accesses will be needed to retrieve the whole record, a probability of (M/K + 1)(M/K)/N that exactly three accesses will be needed, and so on. As N, K, and M vary, the expected number of accesses will vary.

In the table in Appendix L, the breakpoints are given at which both storage strategies lead to the same expected number of accesses. The top row contains the values of K, the column farthest to the left contains values of M, and the remaining columns contain values of N. Keeping K and M fixed, if N is less than the value in the table, then the whole-record storage strategy requires fewer accesses; if N is greater than the value in the matrix, then the residual-record storage strategy requires fewer accesses. It can be seen that, for a given number of occurrences of a record, the larger the record and the smaller the key data element the more advisable it is to store only the accessing information in the record type data set.

An initial implementation restriction in SPIRES II will be that the data of any record type that passes data element values to another record type will be removed to the residual data set.

4.5.2 The Logical Effects of Removal

Figure 22 shows a file set consisting of four record type data sets. Type 1 is the ID records, which are also the goal records. Types 2, 3, and 4 are access records. It should be noted that all access record pointer groups are based on key data element values; in general, whenever access records point to goal records with unremoved data, the pointers are of this sort.

It should also be noted that although Figure 23 shows the data portion of the ID records stored in the residual data set, the removal decision may be made on any or all record types within a file. The decision is, of course, always made according to the algorithm given in section 4.5.1.

4.6 RECORD SPLITTING

It is possible that even though a node pointer group begins in a tree-structured data set it may continue and end in the residual data set. The reason for this is that the number of tree levels is directly proportional to the node size, given some fixed number of nodes. Thus a tree-structured data set may become inefficient to access if large nodes are stored there in their entirety. A maximum node size (a file characteristic) is used to determine whether or not splitting should occur. As much of the record value as possible is stored in the node, and the rest is stored in a residual entry. The node points to the residual data set.

Even if removal has occurred and record values are stored directly as entries in the residual data sets, there are times when more than one residual entry will be needed to store the value. This occurs when the record value is larger than the maximum residual data set entry size. Then the first part of the

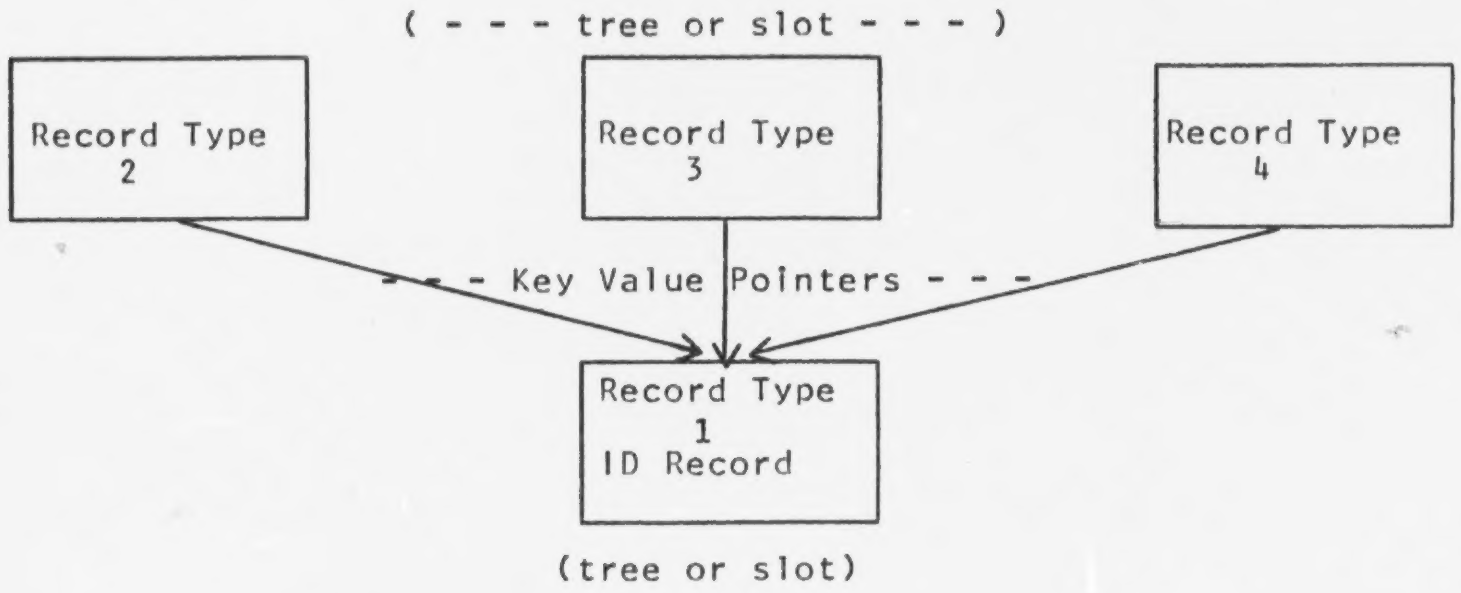

(- - - tree or slot - - -)
Record Type 2
Record Type 3
Record Type 4
- - - Key Value Pointers - - -
Record Type 1 ID Record
(tree or slot)

Figure 22. File Set Without Removal

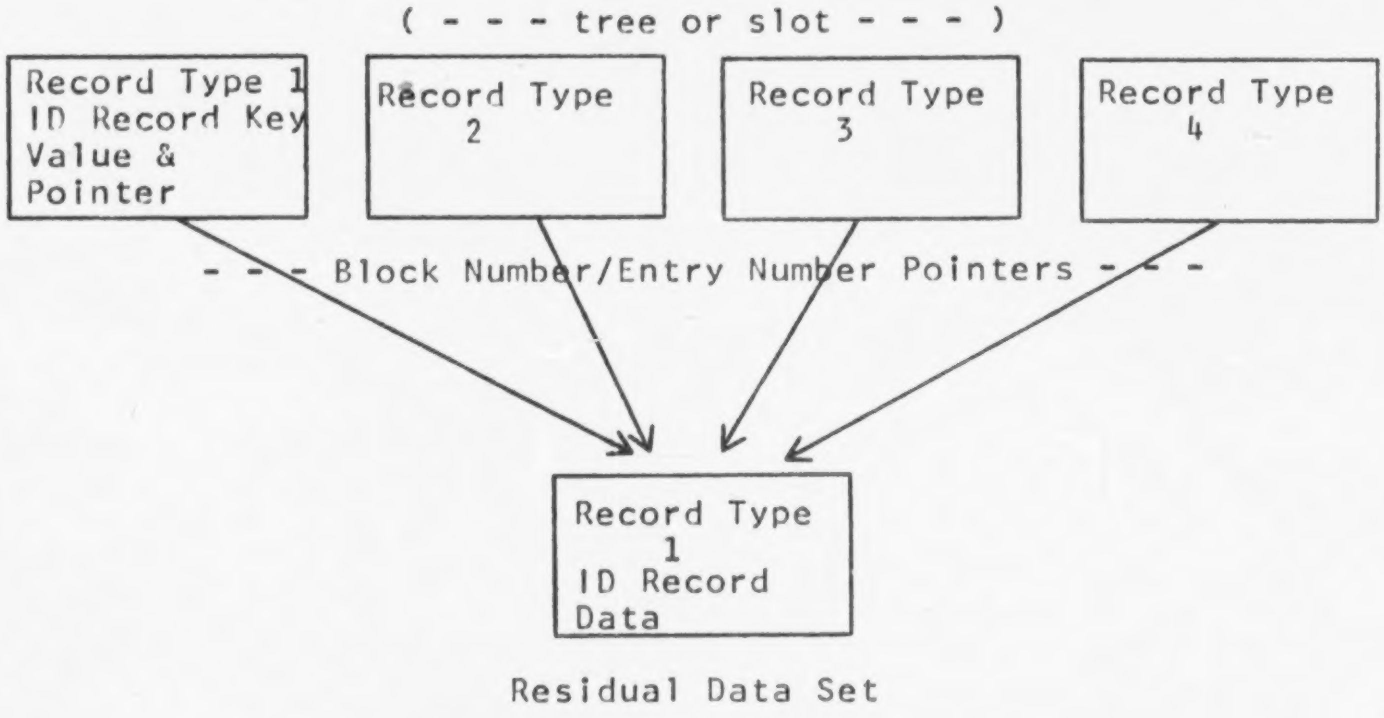

(- - - tree or slot - - -)
Record Type 1 ID Record Key Value & Pointer
Record Type 2
Record Type 3
Record Type 4
- - - Block Number/Entry Number Pointers - - -
Record Type 1 ID Record Data
Residual Data Set

Figure 23. File Set with Removal

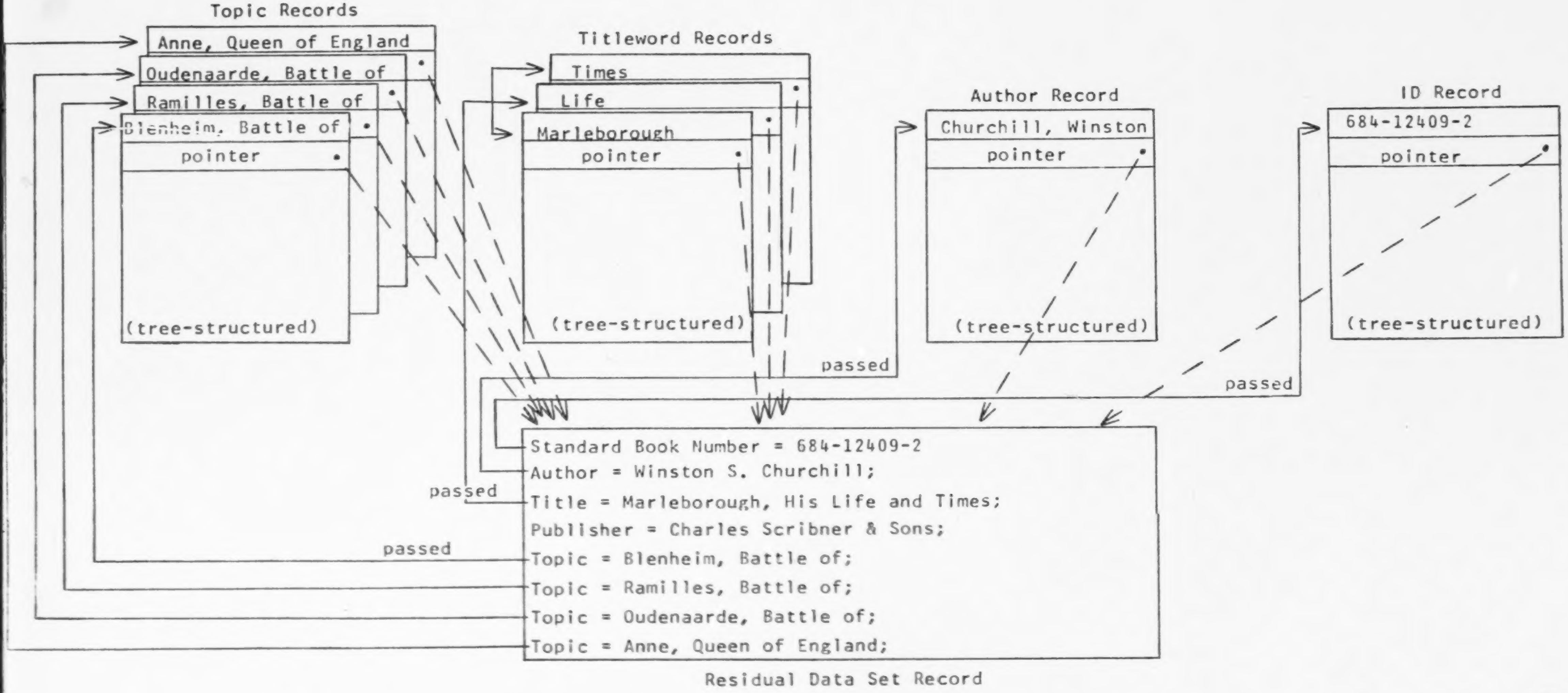

Topic Records
Anne, Queen of England
Oudenaarde, Battle of
Ramilles, Battle of
Blenheim, Battle of
pointer
(tree-structured)
Titleword Records
Times
Life
Marleborough
pointer
(tree-structured)
Author Record
Churchill, Winston
pointer
(tree-structured)
ID Record
684-12409-2
pointer
(tree-structured)
passed
passed
passed
passed
Standard Book Number = 684-12409-2
Author = Winston S. Churchill;
Title = Marleborough, His Life and Times;
Publisher = Charles Scribner & Sons;
Topic = Blenheim, Battle of;
Topic = Ramilles, Battle of;
Topic = Oudenaarde, Battle of;
Topic = Anne, Queen of England;
Residual Data Set Record

Figure 24. An Example of Passing and Removal

record value is stored in one entry with a pointer to another residual entry that contains the rest.

4.7 A SIMPLE ILLUSTRATION OF PASSING AND REMOVAL

In Figure 24 there are four record type data sets: an ID record type, an Author record type, a Titleword record type, and a Topic record type. The data within the ID records is to be removed to the residual data set. The key data element values for the Author, Titleword, and Topic records will be passed from the ID record data in the residual data set.

CHAPTER 5

PHYSICAL FORMATS

5.1 INTRODUCTION AND DEFINITIONS

This chapter describes the formats of the various kinds of user data sets in SPIRES II. The records in tree-structured data sets are called nodes; the records in non-tree-structured data sets are called entries. The record format common to both nodes and entries is given. A system data set of crucial importance, the account number tree, is described. The format of the user master data set is also outlined. From these last two descriptions one learns how the profile is implemented.

5.2 RECORD FORMATS

When the user defines his file he specifies either implicitly or explicitly the record types within the file and any structures within a record type. Internal structures are governed by the same rules governing record types and are, in fact, treated as records within records. Therefore, records and their internal structures are formatted similarly.

A record or an internal record structure is made up of one to three different sections. These are the FIXED REQUIRED section, the REMAINING REQUIRED section, and the OPTIONAL section --in that order.

The FIXED REQUIRED section contains elements of fixed length and fixed occurrence. If there is a fixed-length key data element, it must be the first element of the FIXED REQUIRED section. (Note: A record must have a key data element, but an internal record structure need not have a key data element. Also, a key data element must occur only once.)

Following the FIXED REQUIRED section is the REMAINING REQUIRED section. If there is a variable-length key data element, it must be the first element of the REMAINING REQUIRED section. If there is a possibility of optional elements, an optional element bit mask comes next. Then come the remaining required elements that are either variable in length, variable in occurrence, or both.

Following the REMAINING REQUIRED section comes the OPTIONAL section. Within this section are those elements that need not occur, but which, when they do, may be fixed or variable in length and occurrence. The bits in the optional element bit mask are used to determine if an optional element occurs and, if so, how many other optional elements occurred before it.

Each element in the REMAINING REQUIRED and OPTIONAL sections begins with a TOTAL VALUE LENGTH HEADER (TVLH), as does the optional element bit mask when it occurs.

This format has been chosen because of its simplicity. There is no need for data element pointers; the total lengths preceding elements provide values to be used during record value validity tests. The design favors records that are updated infrequently but accessed often, and that are accessed for all their data element values rather than for a few.

From the above, it is seen that each data element value is either fixed or variable in length; the element may either occur singly or multiply, and it may be required to appear at least once, or may be optional. Figure 25 shows the total value length headers and other control information that must prefix data elements whose occurrence or length is variable.

If the record contains only fixed-length, fixed-occurrence, required data elements, then the record will contain only a fixed required section. If the record contains only required data elements, but some elements may have variable lengths or some elements may occur a variable number of times, then the record will have a remaining required section. If the record can contain optional elements, then the record will have a remaining required section, and an optional section.

Users will be encouraged to define as fixed in length and occurrence the data elements they plan to access and update heavily. They will also be encouraged to define their required elements so that the most heavily accessed are physically stored first and the most heavily modified are stored last. If optional data elements need to be accessed frequently, then they can be defined as required data elements and given some default value when they do not occur. If merely the fact of the occurrence or absence of an optional element is all that is needed, then that information is stored in the easily accessed optional data element bit mask. The optional data elements section should be used for the data elements in a record that fluctuate the most radically.

5.3 FILE BLOCK FORMAT

All physical file blocks are 2,048 bytes in length. Most nodes and entries will be a fraction of that size; a few entries may exceed that size. Therefore, a block management scheme is necessary that accommodates nodes and entries within a block as well as nodes and entries that span blocks. Since physical blocks constitute the basic unit of storage and transfer, they must contain the control information that enables validity checking, damage lockout, and other reliability mechanisms to operate.

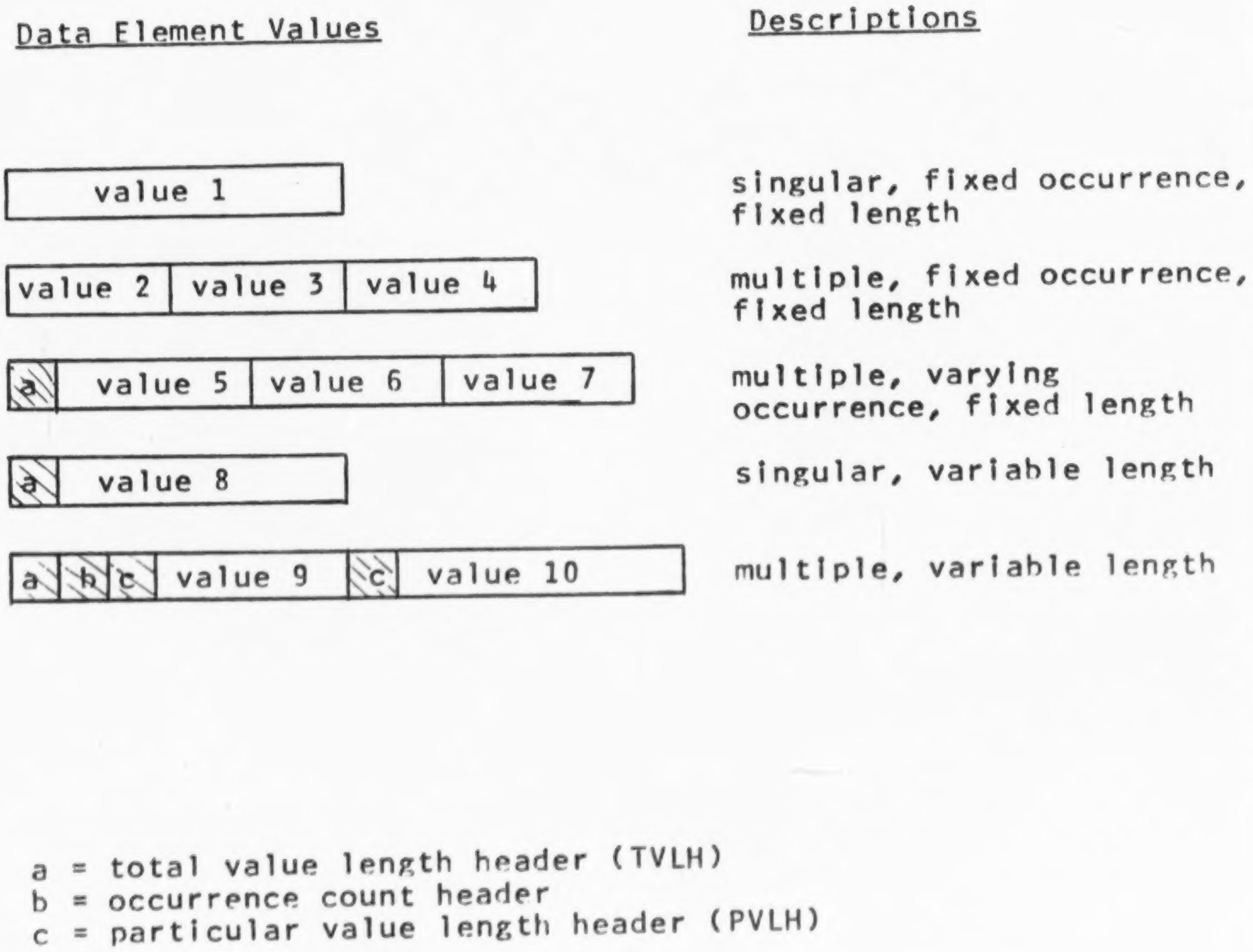

Data Element Values
Descriptions
value 1
singular, fixed occurrence, fixed length
value 2
value 3
value 4
multiple, fixed occurrence, fixed length
a
value 5
value 6
value 7
multiple, varying occurrence, fixed length
a
value 8
singular, variable length
a
b
c
value 9
c
value 10
multiple, variable length
a = total value length header (TVLH)
b = occurrence count header
c = particular value length header (PVLH)

Figure 25. Control Information Appended to Various Data Element Values

The basic file block format for tree data sets is almost the same as the format for non-tree data sets. The two are treated separately below, however, and their few differences are pointed out.

5.3.1 The Tree Data Set Block Format

Regardless of record format, the blocks of all tree data sets have the same format. Block size and maximum node size are system constants. No tree can contain more than 32,767 blocks. Each block begins with a header followed by nodes that build toward the end of the block, and trailers that start from the end of the block and build toward the header information. When there is no more available space between the nodes and the trailers, then the block is full.

There are five data elements in the BLOCK HEADER. The first is four bytes long and contains the date and sequence number of the last transaction that modified the block. The second is the two-byte block number--this is the number of the block relative to the beginning of the tree data set.

The third data element in the header is two bytes long and contains the number of trailers in the block. There is exactly one more trailer than there are nodes in the block, so this data element can also be used to calculate the number of nodes. (One trailer is reserved to locate available space.)

The fourth data element contains the number of branch pointers in the block. Recall from the explanation of tree growth that nonterminal nodes are those that are preceded by a branch pointer, and that all nonterminal nodes in a block alphabetically precede all terminal nodes. When a block is being searched for a key data element value, the branch pointers do not get in the way. When it is found that the key data element value falls between nodes n1 and n2, then the fourth data element of the header is used to determine whether or not node n2 is preceded by a branch pointer. If it is, then that branch pointer is used to access the next block of the tree; if it is not, then there is no node in the tree with this key data element value.

The fifth data element in the header is a two-byte block number. Instead of being the number of this block, it is the number of the preceding block in the tree, i.e., the block that contains the branch pointer to this block. When a block in the tree is examined, this number is matched against the previous block's number to ensure that the tree is functioning properly. If a block of the tree is damaged, the sign bit of the second data element is set to on (-1). This causes the reliability test to fail and prevents the block from being used.

At the very end of each block is a duplicate image of the first data element that contains the date and sequence number of the last transaction that modified the block. Storing duplicate information at the beginning and end of each block makes it possible to identify a half-written block simply by comparing the first and last words of the block.

The TRAILERS, which point to the nodes in the block, build towards the header information in alphabetical order. For example, if the three nodes in the block have key data element values of hot, cold, and warm, then the trailer at the end of the block points to the second node. The trailer before that points to the first node, and the trailer before that points to the third node. Every trailer is four bytes long. The first three bits contain status information about the node pointed to. If the first bit (the "sign bit") of the trailer is set to off (+0), then all of the status bits are set to off. Otherwise, at least one of the remaining status bits is in use. The second bit is set to on when a node has been so damaged that it should not be used. The third bit is set to on when a node has been logically, although not physically, deleted from the tree. The fifth through the sixteenth bits of the trailer contain the size (in number of bytes) of the node. The seventeenth bit is set to on when the node contains only the first segment of the record value. The last 12 bits of the trailer, bits 21 through 32, contain the displacement of the node within the block (i.e., the byte address of the node, where the first byte of the block has a displacement of zero). Bits 4, 18, 19, 20 are not currently used for nodes.

There is always one more trailer than there are nodes. The trailer closest to the nodes contains the location and length of the available space in the block. Its first and third bit are set to on even though all its other status bits are set to off. This is to distinguish it from all other trailers.

Little has to be said about the NODES and BRANCH (block) POINTERS in a block. If the node is nonterminal, it is preceded by a two-byte branch pointer that is the block number of the next block down in the tree. Trailers point to the beginning of the nodes rather than to the beginning of branch pointers, and the node length given in a trailer does not include the length of the branch pointer. If a trailer indicates that a node does not contain the entire record value, then the last four bytes of the node contain the residual block number and the relative trailer location within that block where the next segment can be found. If there is one more branch pointer in the block than there are nodes, then the final branch pointer is stored before the available space. Figure 26 is a graphic representation of the tree block format, and Figure 27 is a table showing the lengths and displacements of the various data elements in a tree block, as well as a non-tree block.

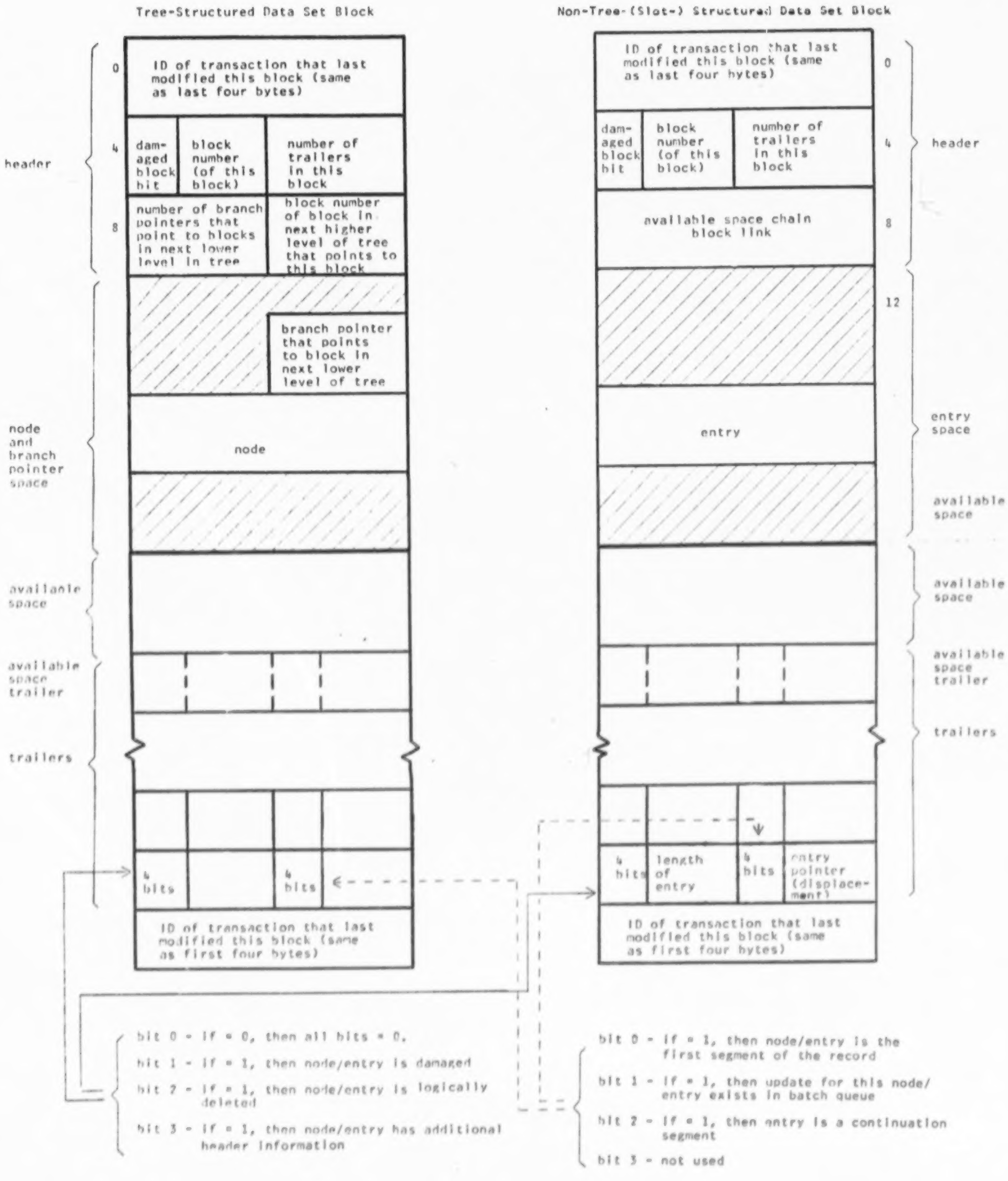
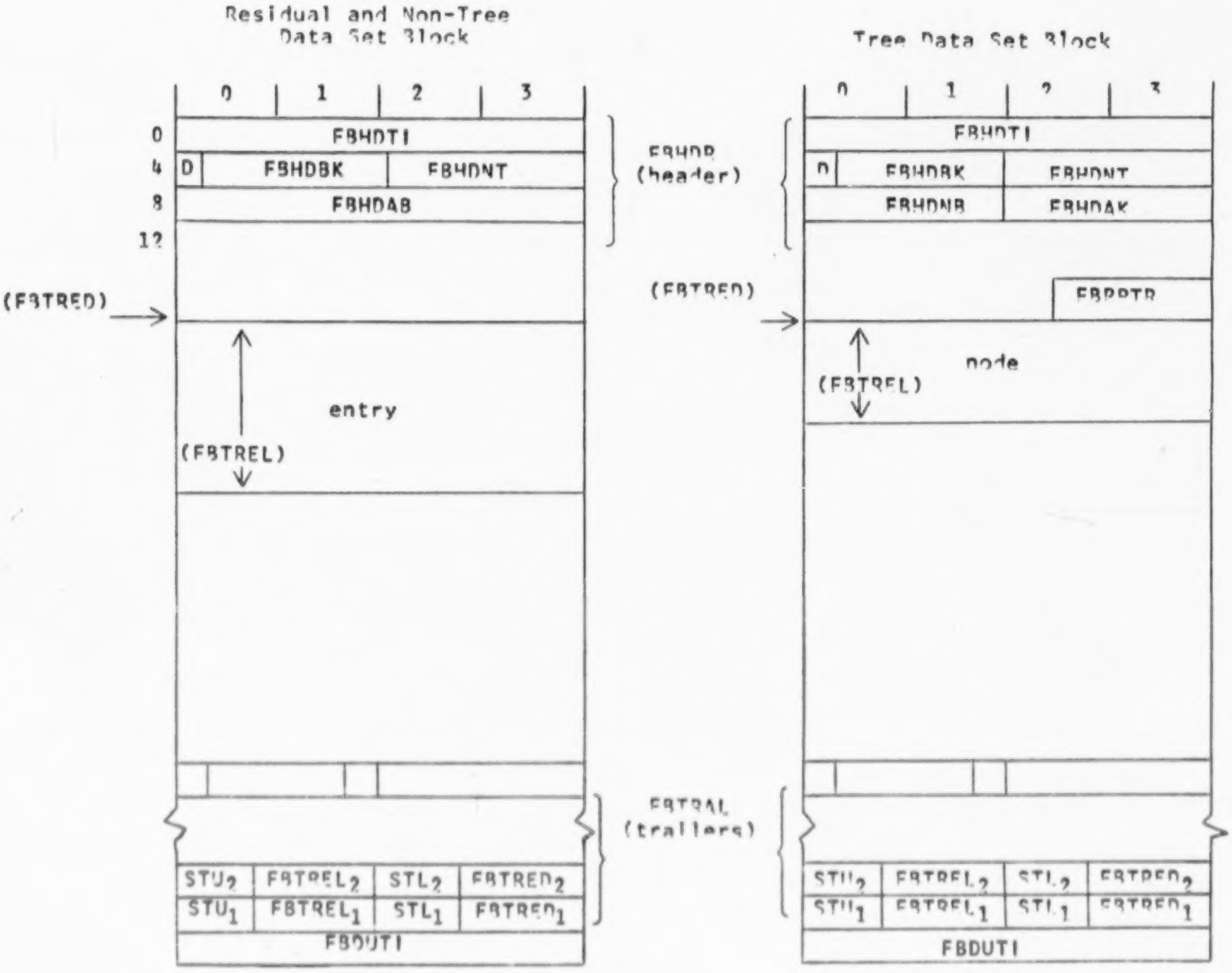

Tree-Structured Data Set Block
Non-Tree-(Slot-) Structured Data Set Block
header
0
ID of transaction that last modified this block (same as last four bytes)
4
damaged block bit
block number (of this block)
number of trailers in this block
8
number of branch pointers that point to blocks in next lower level in tree
block number of block in next higher level of tree that points to this block
available space chain block link
12
branch pointer that points to block in next lower level of tree
node and branch pointer space
node
entry space
entry
available space
available space trailer
trailers
4 bits
length of entry
entry pointer (displacement)
ID of transaction that last modified this block (same as first four bytes)
bit 0 - if = 0, then all bits = 0.
bit 1 - if = 1, then node/entry is damaged
bit 2 - if = 1, then node/entry is logically deleted
bit 3 - if = 1, then node/entry has additional header information
bit 0 - if = 1, then node/entry is the first segment of the record
bit 1 - if = 1, then update for this node/entry exists in batch queue
bit 2 - if = 1, then entry is a continuation segment
bit 3 - not used

Figure 26. File Block Formats


Residual and Non-Tree
Data Set Block
Tree Data Set Block
0
1
2
3
0
4
8
12
FBHDTI
D
FBHDBK
FBHDNT
FBHDAB
FBHDR
(header)
FBHDNB
FBHDAK
(FBTRED)
FBRPTR
entry
node
(FBTREL)
FBTRAL
(trailers)
STU_2
$FBTREL_2$
STL_2
$FBTRED_2$
STU_1
$FBTREL_1$
STL_1
$FBTRED_1$
FBDUTI

FIELD NAME	M	DISP.	B	LENGTH	B	D	FIELD DESCRIPTION
FBHDR		0-fw		12			Block Header
--FBHDTI		0		4			Last transaction ID which modified this block.
--FBHDD		4		1	x		Damaged Block Indicator
--FBHDBK		4		2			This field holds the block number of this block (mod 32768)
--FBHDNT-(nt)		6		2			Number of trailers in the block
--FBHDAB		8		4			Available space block linkage
--FBHDNB		8		2			Index Block - Number of branch pointers.
--FBHDAK		10		2			Index block - The block number through which this block is accessed.

Figure 27. File Block Structures

FIELD NAME	M	DISP.	B	LENGTH	B	D	FIELD DESCRIPTION
FBDUTI		blksiz - 4		4			Duplicate transaction ID for damaged block checking
FBTRAL	nt	Blksiz -4-n*4		4			Trailer - n=trailer number (references block entry)
--FBTRSTU		0	X	4	X		Upper status bits
						1	0000; indicates normal entry (sign bit= 0, Then all are zero)
						2	11XX, the entry is damaged
						3	1X1X, the entry is logically deleted. (if length field is zero, the entry is physically deleted)
						4	1XX1, the entry has header information which further defines the entry contents.
--FBTREL		4	X	12	X		Length (bytes) of entry (includes segment pointer)
--FBTRSTL		16	X	4	X		Lower status bits
						1	1XXX, the entry is an initial segment. (within this data set)
						2	X1XX, the entry has been updated and now exists in the Batch Queue. (if FBTRSTU = 1X00) Or the entry has been logically deleted and is referenced in the Delete Queue (if FBSRSTU = 1X1).
						3	XX1X, the entry is a continuation segment (Residual Only) note: if entry is a segment there will always be an fbenpt. The last segment in the chain locates the initial seg.
--FBTRED		20	X	12	X		Displacement (bytes) of entry within the block.
FBENPT		--		4			Entry Pointer
						1	Index overflow - Locate overflow segment in residual.
						2	Residual overflow - Locates overflow segment in residual.
						3	Batch Queue - Delete Queue Locator. Locates entry in BATCH or DELETE QUEUE.
--FBEBLK		0		3			Block number of the Data Set holding the entry.
--FBENUM		3		1			Trailer number in block which locates the entry.
FBRPTR		--		2			Branch Block Pointer in an Index Block. Locates lower level block in this data set.

Figure 27 continued

5.3.2 The Non-Tree Data Set Block Format

"Non-tree data sets" is a term that may be applied to slot-structured data sets, residual data sets, the master data set, and some system data sets.

A block of the non-tree set is almost identical to a tree block. Every non-tree block begins with header information, entries that build toward the end of the block, and trailers that start near the end of the block and build towards the header information. When the entries and trailer meet, the block is full. The first data element of the non-tree block header is a four-byte transaction ID; the second data element is the block number; the third data element is the number of trailers--all the same as in the tree record. The fourth data element is different from the one in the tree block header; it is the available space chain pointer. For a detailed explanation of this data element, see the discussion of the available space scheme in section 5.4.1.

The non-tree block trailers are almost identical to the tree block trailers except for the order in which they are arranged. When an entry is assigned to a residual block, trailers are scanned starting from the end of the block until a "free" trailer is found. A free trailer is one that corresponds to some previously deleted entry, whose sign bit deleted entry status bit are set to on, and whose length field is zero. If the available space trailer is the first free trailer found, then a new available space trailer is created. Once a trailer has been assigned to an entry, it cannot be released until the entry has either been physically deleted or else moved to another block during cleanup.

The first three status bits function in exactly the same way as do the tree block status bits. The fifth through sixteenth bits contain the size of the entry. If either bit 17 or bit 19 is on, then the last four bytes of the entry contain a pointer to the block and the relative trailer location of the next segment of the entry. When bit 19 is set to on, the entry is a continuation of either a node or another entry. Bits 21 through 32 contain the displacement of the entry within the block.

Figure 26 may be studied for a pictorial representation of the non-tree block format, and Figure 27 for the lengths and displacements of the various elements.

5.4 RESIDUAL BLOCK FORMATS

The components of a residual data set are shown in Figure 28. The data content block format has already been described in section 5.3.2 (The non-tree data set block format). Blocks 0, 1, and 2, however, have special uses and are described below.

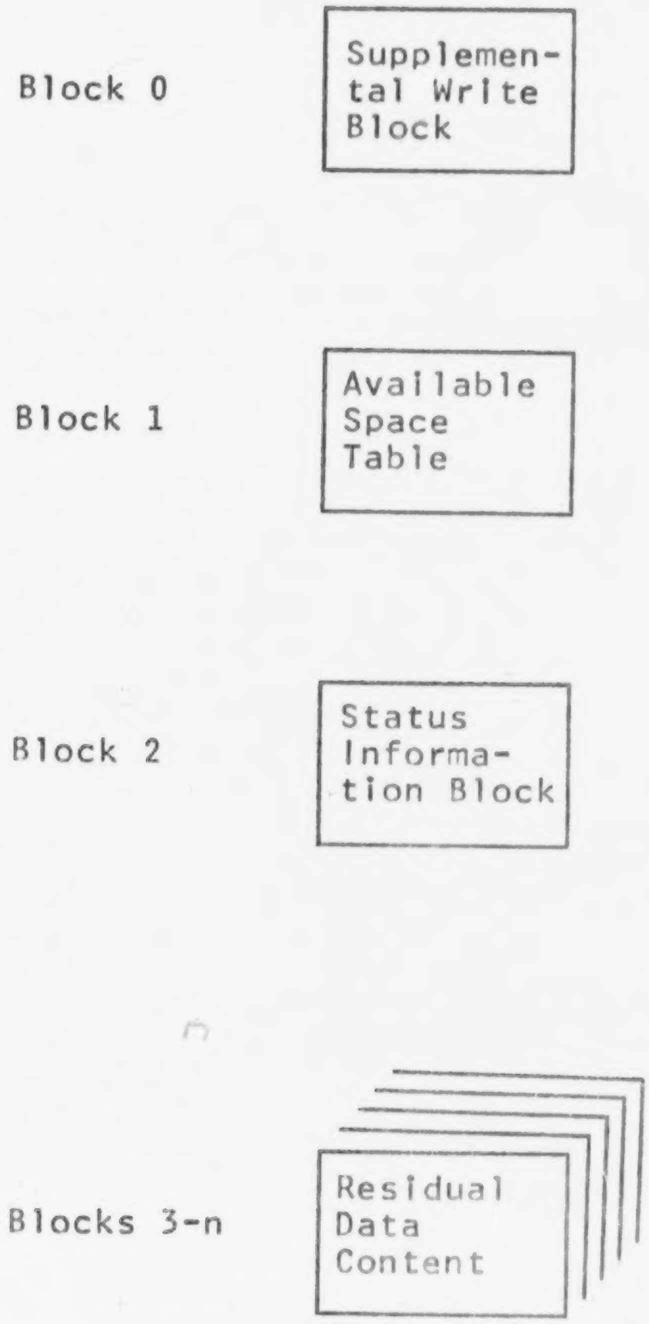

Block 0
Supplemental Write Block
Block 1
Available Space Table
Block 2
Status Information Block
Blocks 3-n
Residual Data Content

Figure 28. Residual Data Set Organization

5.4.1 The Available Space Table

There are two kinds of empty space in the residual data set: the contiguous set of empty blocks at the end, which have never contained data, and the "holes" in the blocks that do contain data. These holes result from the imprecise fitting of entries into blocks during file building, from the deletion of whole blocks, and from the replacement of records by updated versions shorter in length.

When a requirement for space arises during an update run, the strategy will always be to find the smallest segment of available space that fulfills the requirement. Such a strategy minimizes the amount of space broken down into segments too small for use.

Figure 29 is a graphic representation of the mechanism for locating a required amount of space; it includes the format of the available space table. Block 1 in the figure consists of the standard non-tree block header and trailer and a single record consisting of an array of four-byte record pointers, 506 data elements in all. These pointers form the heads of a series of available space chains. These chains are lists of residual blocks; all the blocks listed on a particular chain (usually) have the same amount of available space.

Requests for available space for record additions are handled in the following way. The number of bytes required is rounded up to the next multiple of four. The resulting number is used as an index to the proper pointer in the available space table. This pointer either contains the block number of the first block in a logical chain of blocks, all supposedly (the equivocation is explained below) containing the required amount of available space, or it is set to off. If the pointer is set to off, then no blocks exist that contain the required amount of space, and pointers to increasingly larger amounts of space are examined until a nonzero pointer is found. Once a nonzero pointer is located, the first block in the chain is accessed. A check is made to see if the available space in the block equals the amount assumed for the chain. If not, the block is placed at the head of the correct block chain, and the next block in the chain is read and similarly examined. When a block is encountered in the chain that contains the correct amount, the space is allocated and used. If there is still space available in the block, the block is still pointed to as part of the same chain, unless it happens to be the first block in the chain. In that case, the block is "relocated" in the correct chain.

Deletions and replacements in the space chain are handled similarly. Even though the available space in the block has grown larger or smaller, the block is left on the same chain, unless it happens to be the first block of the chain. The reasoning behind this algorithm is as follows. To make

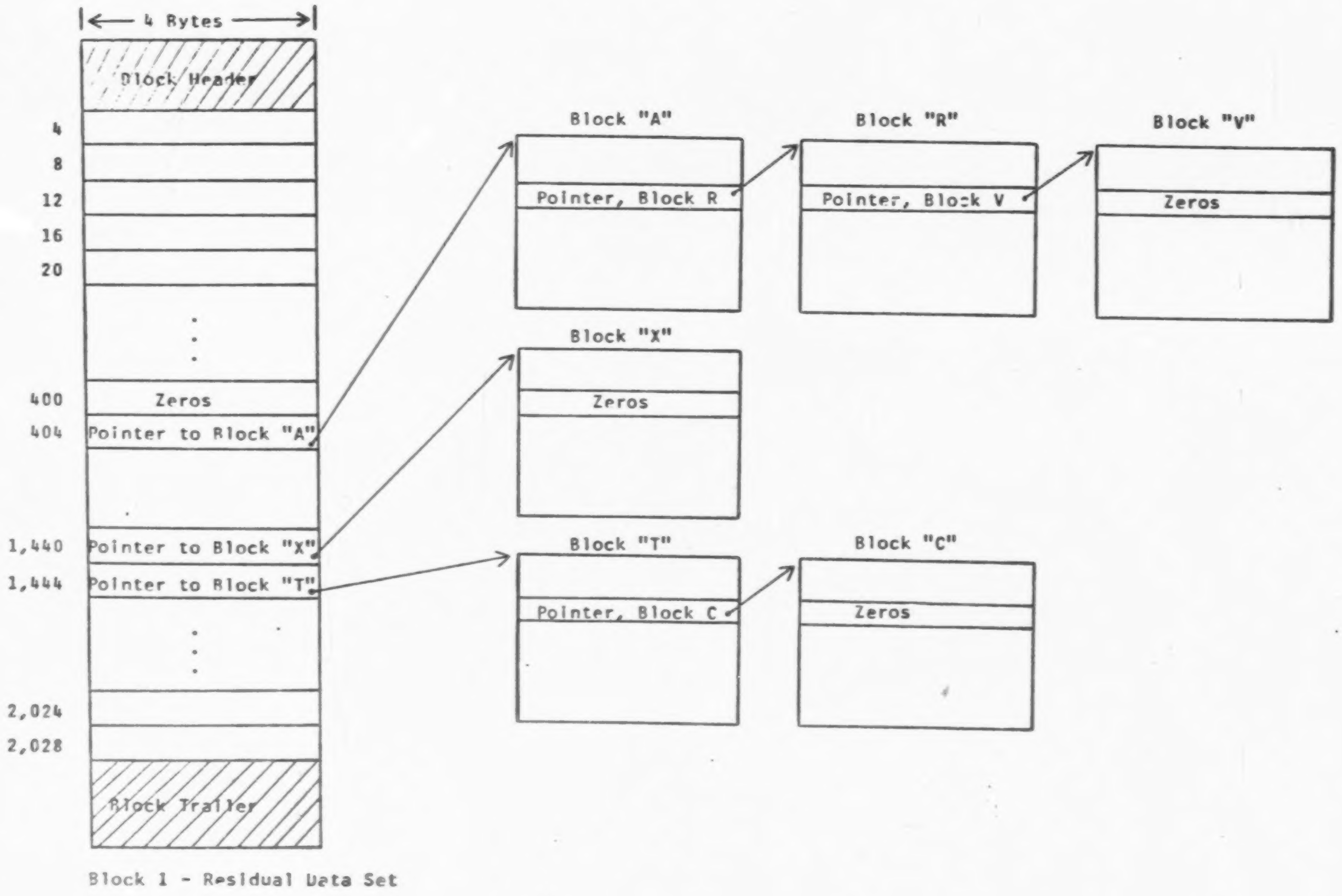

4 Bytes
Block Header
4
8
12
16
20
400
Zeros
404
Pointer to Block "A"
1,440
Pointer to Block "X"
1,444
Pointer to Block "T"
2,024
2,028
Block Trailer
Block 1 - Residual Data Set
Block "A"
Pointer, Block R
Block "R"
Pointer, Block V
Block "V"
Zeros
Block "X"
Zeros
Block "T"
Pointer, Block C
Block "C"
Zeros

Figure 29. The Available Space Mechanism

on-the-spot chain corrections, one must have access to the preceding block in the chain. Since two-way pointers cannot be maintained easily, this means beginning at the head of the chain and following it down until the preceding block is reached. In contrast, chain corrections at the head of the chain can be made without accessing. Therefore, blocks with incorrect amounts of space for their chain will be left to "pop up" to the head before they are switched to a new chain.

5.4.2 Supplemental Write Blocks

Any on-line writes to the file will be done twice: once to the supplemental write block, and once to the block being worked with. When the system is restarted after a crash, but before users are allowed to log on, a small module will be called to perform the following check on every supplemental write block in the system.

> Does the first word of the block match the last word of the block? If not, the supplemental write was not successfully completed, and the update sequence (reserve, read, update in core, write, release) failed during a critical phase but cannot be restarted. If the first and last word agree, then perform the next check.
>
> Does the content of the supplemental write block equal that of the block being double-written? If so, the last update operation was completed successfully. If not, re-initiate the update sequence using the contents of the supplemental write block.

5.4.3 Status Information Block

This block contains a pointer to the beginning of contiguous empty blocks at the end of the data set, and the file update table. This table contains the julian date of the last dump made and an array containing the last seven julian dates on which either a batch update run or a deferred update transaction occurred. The remainder of block two is unused at present.

5.5 THE ACCOUNT NUMBER TREE

The account number tree is the primary mechanism for governing access to SPIRES files. When a user logs onto the system, his account number is used as a search key to locate a node in the account number tree. The node that corresponds to his account number contains a series of profiles that may be used only by holders of that account number. During his session at the terminal, the user may only work within one of these permissible profiles. A profile (see section 3.4) outlines a set of access data sets and a goal data set, and defines the data elements that may be used as search arguments, that may be changed, and that may be displayed.

5.5.1 Class Privileges

Users who have the same privileges in accessing and updating certain files belong to the same user class (it may happen that a user class consists of only one user). Each class is identified by a unique account number. Each user class is entitled to use a file in the ways defined by some profile. A profile (see section 3.4) defines a set of access data sets and a goal data set, and lists the data elements that may be used as search arguments, that may be changed, or that may be displayed. A class may work within one or several profile sets, and these profile sets may apply to more than one file set and more than one user class.

5.5.2 Sharing Profiles Among Accounts

As Figure 30 illustrates, more than one account number may be given access to the same profile. Only profiles restricted to a single account number are actually stored with that number. All other profiles are divided into discrete sets, with each set assigned a "pseudo-account number." These pseudo-account numbers are stored with each account number that has access to the set of profiles. In Figure 30, both accounts B001 and B002 have access to profile XYZ, but only B001 has access to profile WW1.

5.5.3 The Format of the Account Number Record

Figure 31 shows the format of the account number record. This record contains definitions for all the profiles that a user class may operate under and pointers to pseudo-account records that contain definitions for the profiles the user class shares with other user classes.

A profile definition consists of the following elements.

The record number of the goal record for this profile.

The password algorithm code, which at the present stage of SPIRES system design is not being used.

The build flag, a one-byte field that indicates whether the build mask is used only for displaying records or for both building and displaying records.

The profile name, a field that must be matched with the name given in the SELECT <name> command.

The file name of the file to which the profile applies.

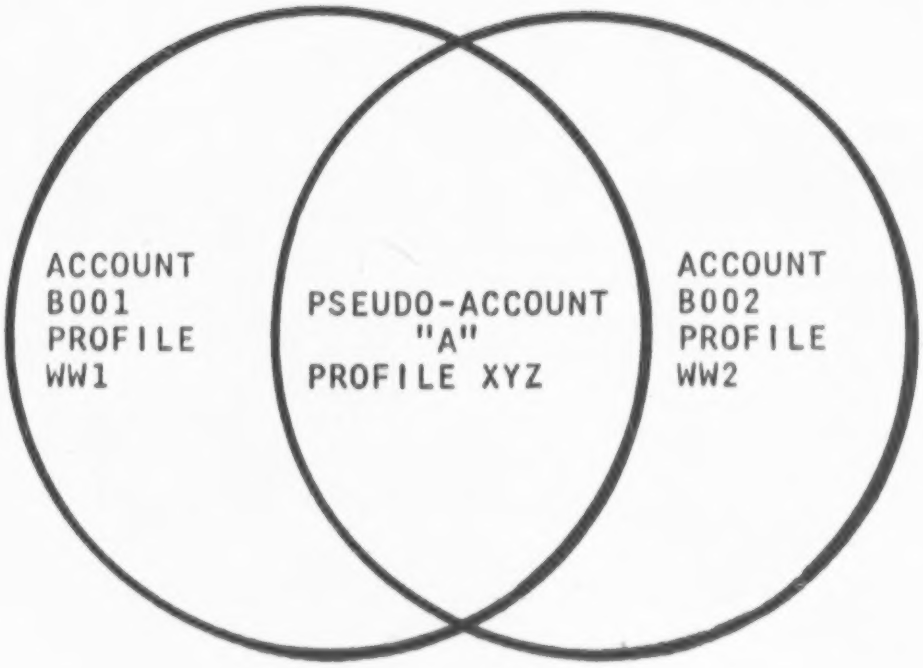
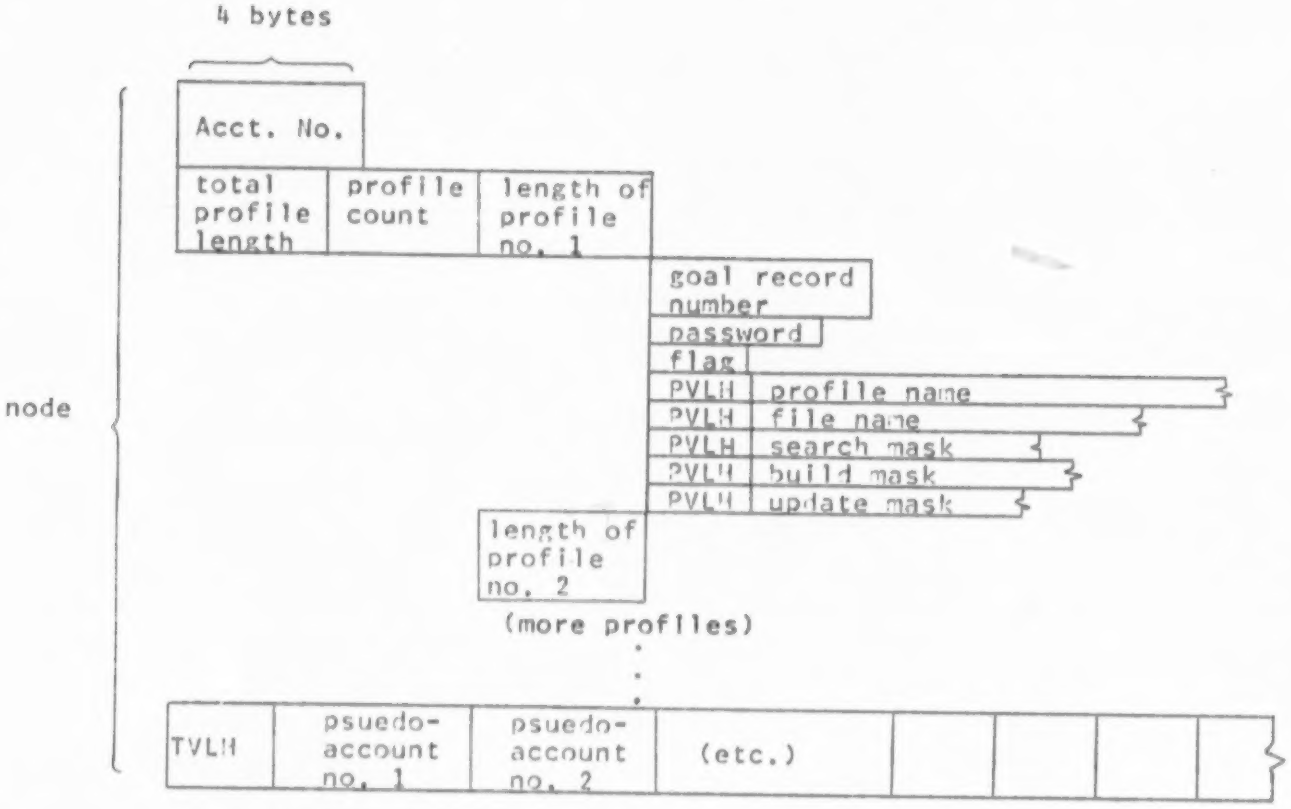

ACCOUNT
B001
PROFILE
WW1
PSEUDO-ACCOUNT
"A"
PROFILE XYZ
ACCOUNT
B002
PROFILE
WW2

Figure 30. Venn Diagram of Account Numbers and Psuedo-Account Numbers


4 bytes
node
Acct. No.
total profile length
profile count
length of profile no. 1
goal record number
password
flag
PVLH
profile name
PVLH
file name
PVLH
search mask
PVLH
build mask
PVLH
update mask
length of profile no. 2
(more profiles)
TVLH
psuedo-account no. 1
psuedo-account no. 2
(etc.)

(indentation shows hierarchy)

Figure 31. Account Number Tree

The search mask. Data element entries in the search characteristics table may be set in such a manner as to cause bits in the search mask to be tested. "1" indicates that the user may name the element in a search command; "0" indicates that he may not.

The build mask. Data element entries in the build characteristics table may be set in such a manner as to cause bits in the build mask to be tested. "1" indicates that the user may display records containing this data element (or this form of a data element); "0" indicates that he may not. If the build flag is on, the user may build records containing the data element.

The update mask. Data element entries in the build characteristics table may be set in such a manner as to cause bits in the update mask to be tested. "1" indicates that the user may change the value of the data element; "0" indicates that he may not.

5.5.4 The Organization of the Account Number Tree

The account number tree is a system file stored under the system account number with the filename "ACCTREE." Besides a master and a residual data set, it has a single record type, that described in section 5.5.3. This record data set is tree structured on the key data element "ACCOUNT NUMBER." Removal of data to the residual data set is unlikely, at least in the beginning. Splitting will occur, however, in cases where the maximum node size is exceeded by an account number with a large number of profiles.

5.6 THE USER MASTER DATA SET FORMAT

The master data set contains record format, build, and search characteristics for all the record types in the file. The contents of the master data set may be categorized as follows (see Figure 32).

The file master structure, which contains some general file information, as well as pointers to the build and search characteristics.

The record format structure, which contains a record format definition for each record type in the file.

The build and search characteristics, which contain the definitions and rules for building, updating, outputting, and searching each record type.

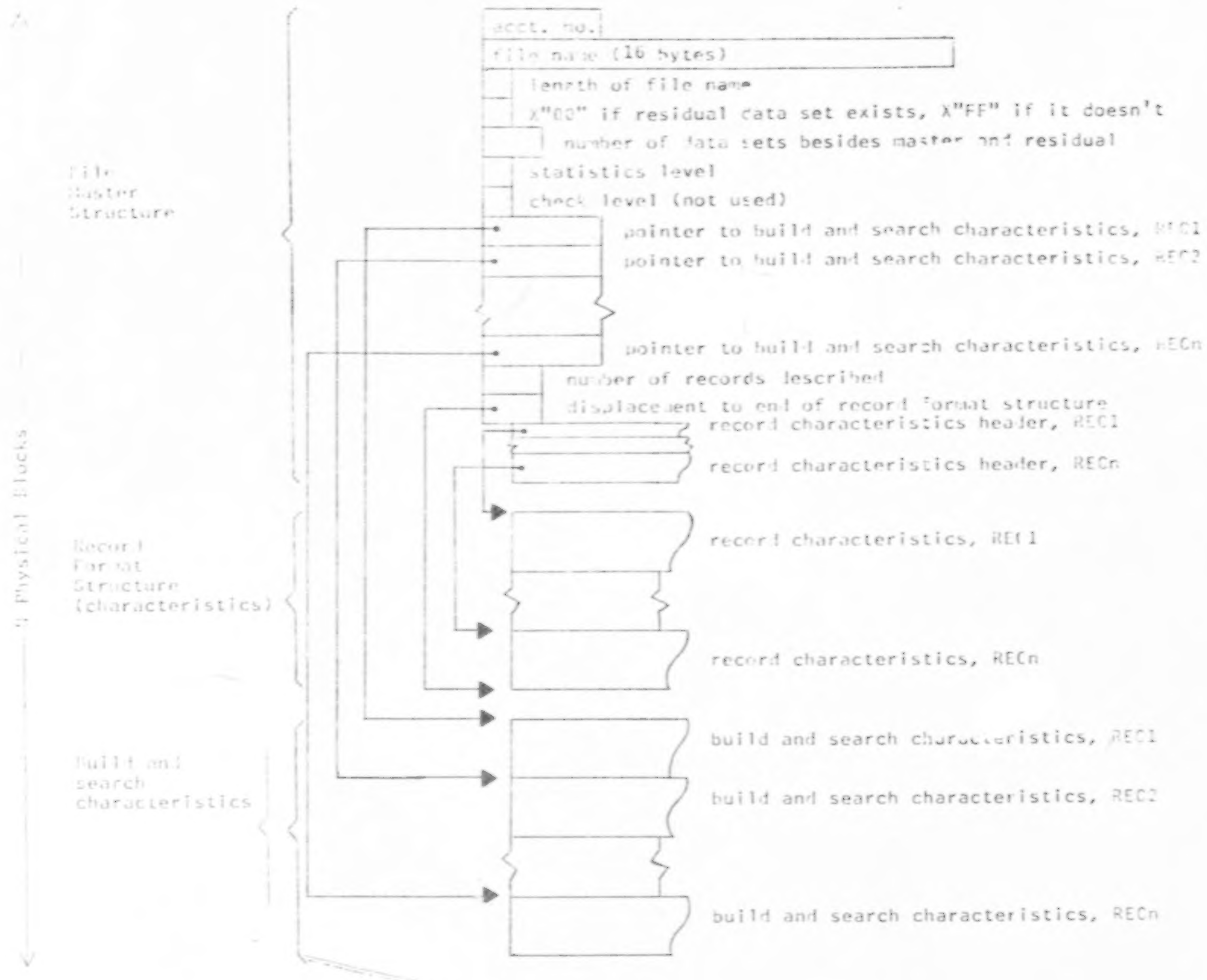

n Physical Blocks
File Master Structure
Record Format Structure (characteristics)
Build and search characteristics
acct. no.
file name (16 bytes)
length of file name
X"00" if residual data set exists, X"FF" if it doesn't
number of data sets besides master and residual
statistics level
check level (not used)
pointer to build and search characteristics, REC1
pointer to build and search characteristics, REC2
pointer to build and search characteristics, RECn
number of records described
displacement to end of record format structure
record characteristics header, REC1
record characteristics header, RECn
record characteristics, REC1
record characteristics, RECn
build and search characteristics, REC1
build and search characteristics, REC2
build and search characteristics, RECn

Figure 32. Organization of the Master Data Set

The record format, build, and search characteristics are spread across an indeterminate number of blocks in the master data set, starting at block 0. The following three sections describe these collections of characteristics in greater detail.

5.6.1 The Record Characteristics

Each record, as well as each internal record structure, may have up to three sections: the fixed required section, the remaining required section, and the optional section. The record characteristics (see Figure 33) consist of one set of characteristics describing all data elements at the record level and one set of characteristics for each internal structure in the record. The format rules for the characteristics themselves do not differ at the record and internal structure levels.

For each record (or internal structure) there will be a series of counts giving the total number of data elements in the record (or in the structure), and for each data element there will be a one-byte code together with a data element length. The code will show where in the record the element resides and what data type it is (character string, internal structure, or record pointer). The data element length applies only to fixed-length data elements, and is left at zero if the element is variable. Following the codes and lengths is a table of halfwords that contain the displacements (from the beginning of the record) of each fixed required data element.

5.6.2 The Build Characteristics

Build characteristics are used to transform a record from external (user-readable) format to internal (machine-readable) format during the file update process, and from internal to external format during the display process. These characteristics are usually used in conjunction with the record characteristics; there is one set of build characteristics per record type. Figure 34 shows the layout of the build characteristics; the components are described below.

> INPUT MNEMONIC DICTIONARY. There are as many entries in the input mnemonic dictionary as there are unique data element mnemonics that may be used for external format input records. There are separate entries for alternative mnemonics. The length of each mnemonic is given (the limit is 16 bytes) and a displacement in the packed character table showing where the actual mnemonic string can be found. Following the displacement is the structure element number (see section 3.5) of the data element that corresponds to the mnemonic.
>
> PACKED CHARACTER TABLE. This is an amalgamation of all the mnemonic character strings.

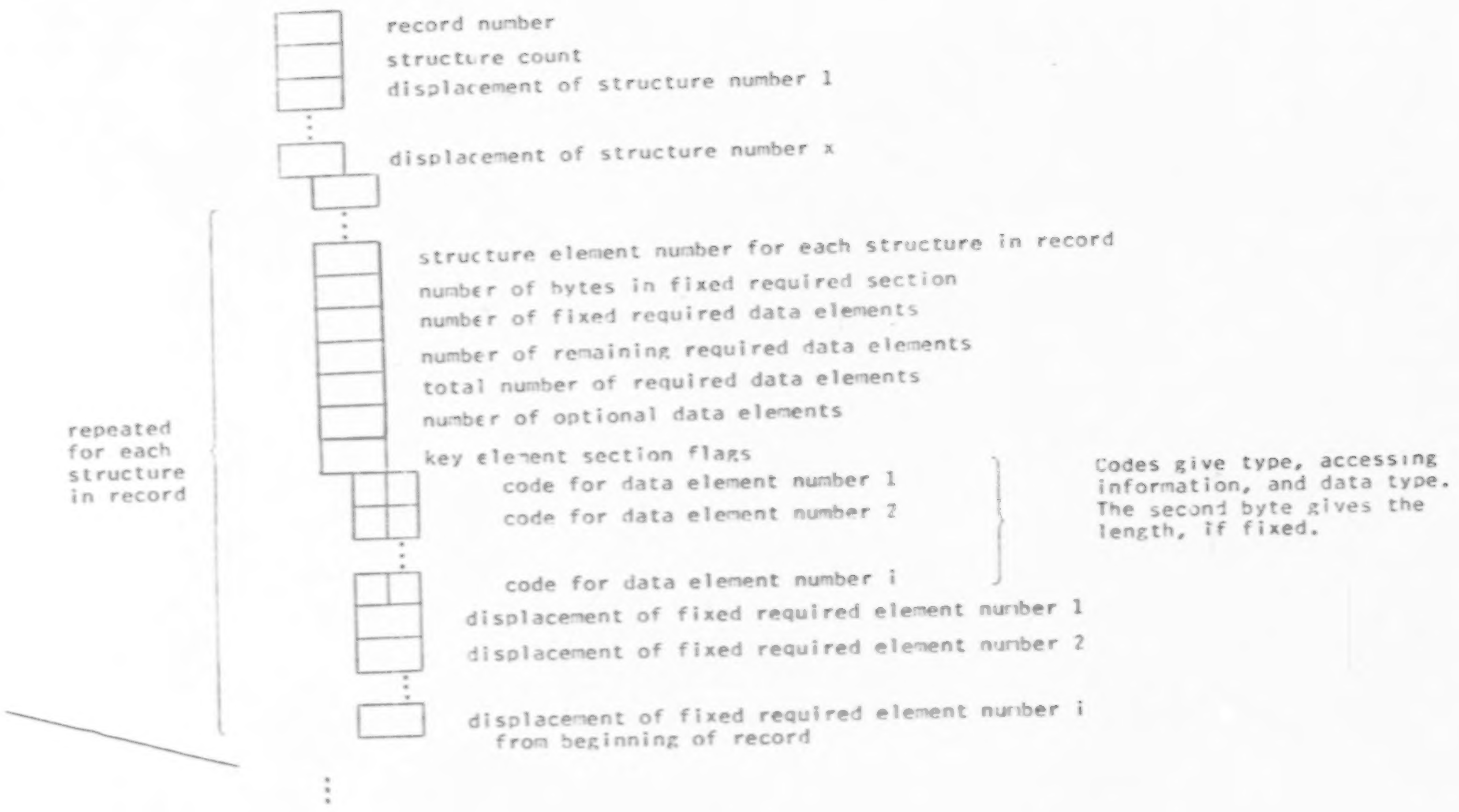

record number
structure count
displacement of structure number 1
displacement of structure number x
structure element number for each structure in record
number of bytes in fixed required section
number of fixed required data elements
number of remaining required data elements
total number of required data elements
number of optional data elements
key element section flags
code for data element number 1
code for data element number 2
code for data element number i
Codes give type, accessing information, and data type. The second byte gives the length, if fixed.
displacement of fixed required element number 1
displacement of fixed required element number 2
displacement of fixed required element number i from beginning of record
repeated for each structure in record

Figure 33. Record Characteristics

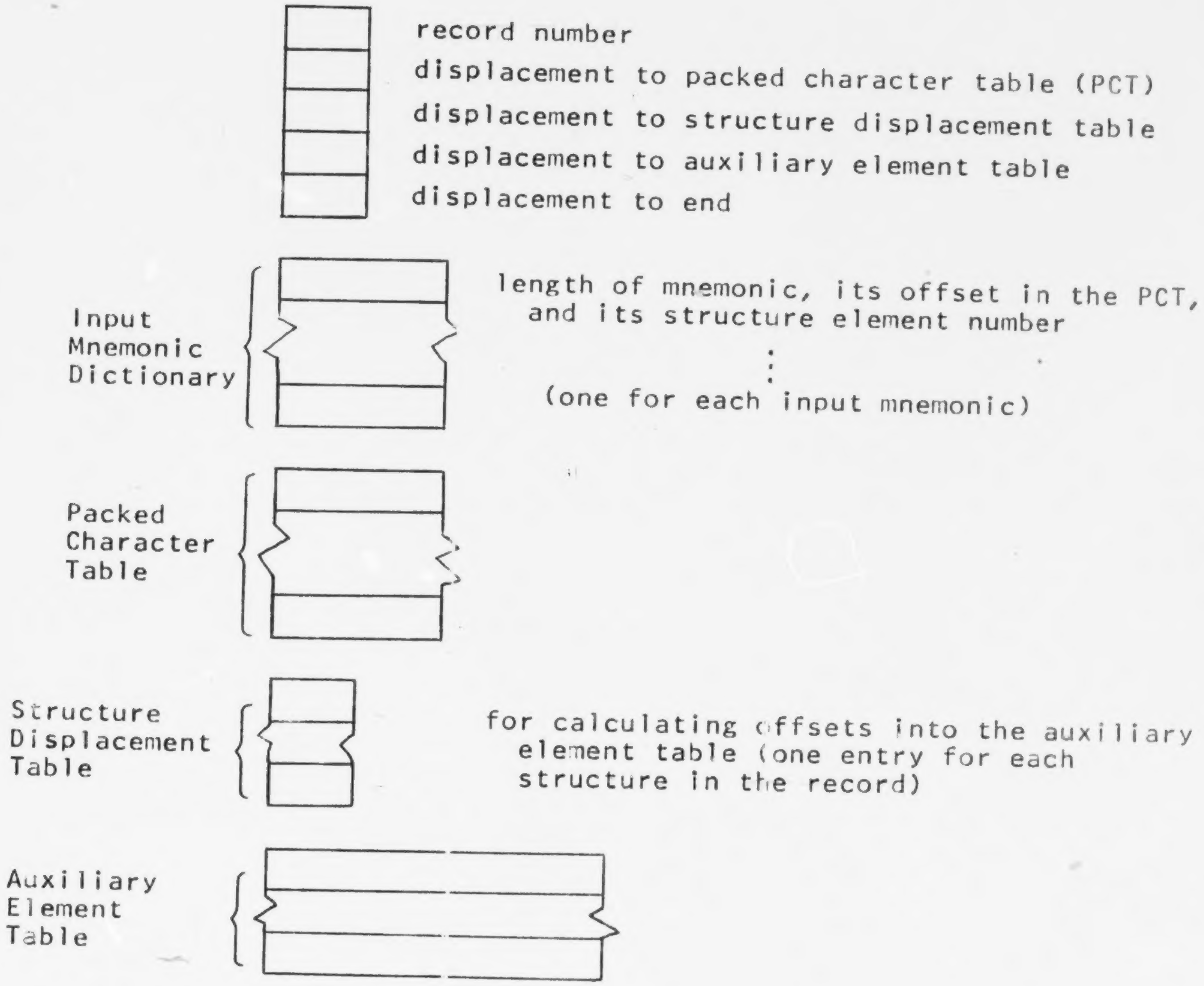

record number
displacement to packed character table (PCT)
displacement to structure displacement table
displacement to auxiliary element table
displacement to end
Input Mnemonic Dictionary
length of mnemonic, its offset in the PCT, and its structure element number
(one for each input mnemonic)
Packed Character Table
Structure Displacement Table
for calculating offsets into the auxiliary element table (one entry for each structure in the record)
Auxiliary Element Table

Figure 34. Build Characteristics

STRUCTURE DISPLACEMENT TABLE. There are as many entries in the structure displacement table (SDT) as there are unique internal structures in the record. The table is accessed by using the structure number as an index. The entries in the table consist of displacements that are used to access the auxiliary element table (AET) in the following way:

AET offset = contents of SDT (structure no.)
+ structure element no.

AUXILIARY ELEMENT TABLE. Contains one entry per data element. The most important part of each entry is the condition byte, which contains the bit number in the profile bit masks that corresponds to the data element. Each entry also contains references to both input and output processing rules, the length of the data element mnemonic, and the mnemonic's displacement in the packed character table.

5.6.3 The Search Characteristics

Search characteristics are only meaningful for goal records. (Search characteristics for record types used only in accessing are essentially null.) Each goal record has one or more data elements that may be given in search commands. These elements may be classified in one of the following three ways:

The data element value is passed to an accessing record.

The data element is a qualifier.

The data element value has not been passed, and it is not the unique key data element of the goal record.

For every such data element within a goal record, there is the following corresponding data in the search characteristics (see Figure 35).

MNEMONIC LENGTH AND DISPLACEMENT. This is the length of the search mnemonic that corresponds to the data element and its location in the packed character table.

CONDITION BYTE. This byte refers to a bit in the search bit mask of the profile. If it is on, the user may use the element to search. If it is off, he may not.

TYPE. This byte classifies the search data element into one of the three classifications already mentioned. The contents of this byte vary with the classification, and determine the contents of the search descriptor.

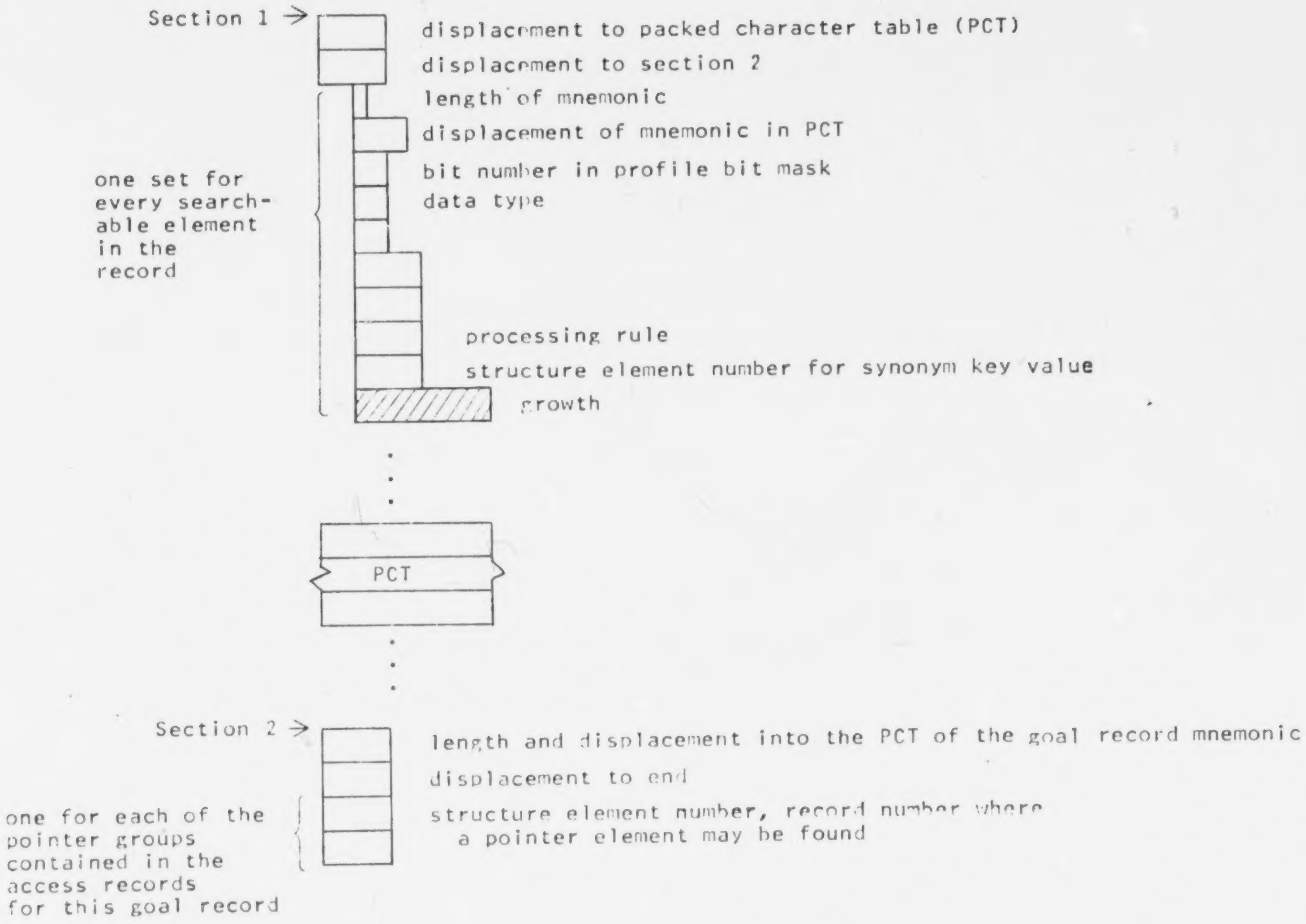

Section 1 →
displacement to packed character table (PCT)
displacement to section 2
length of mnemonic
displacement of mnemonic in PCT
bit number in profile bit mask
data type
one set for every searchable element in the record
processing rule
structure element number for synonym key value
growth
PCT
Section 2 →
length and displacement into the PCT of the goal record mnemonic
displacement to end
structure element number, record number where a pointer element may be found
one for each of the pointer groups contained in the access records for this goal record

Figure 35. Search Characteristics

SEARCH DESCRIPTOR. If the search command data element value is passed to an accessing record, then these four bytes contain the record number of the record to which the value has been passed, and the structure element number of the pointer group within the accessing record.

If the search command data element is a qualifier, then the format is still the same.

If the search command data element has not been passed to an accessing record and it is not the unique key data element of the goal record, then these four bytes contain two structure and element numbers--one for the qualifier package in the goal record, and one for the search data element in the goal record. The data elements in this category are examined by a linear search module.

PROCESS RULE. References the processing rule used to place the data element into standard form for storage. This field is zero if there is no rule.

SYNONYM REFERENCE. This structure element number refers to the field in the accessing record that contains a key data element value pointer to the next record of the same type in a series of synonymous key data element values.

CHAPTER 6

IMPLEMENTATION OF THE SPIRES II ACCESS METHOD

6.1 INTRODUCTION

The concepts underlying the SPIRES file structure, the organization of the data, and the format of the file structure have been treated at length in chapters 3, 4, and 5. This chapter describes the modules that manipulate the structure. The SPIRES II access method is a collection of modules and subroutines that are used singly or in combination to add, change, replace, or delete data in a SPIRES II file.

If the access method routines are being used as part of the on-line system, they reside in core just below SEMANT (see section 2.5) and make calls on the ORVYL interface routines for supervisor services like input and output, file attaching, and so on. If the access method routines are being used by batch programs running under O/S, the calling sequence will still be the same, but I/O requests will be directed to Virtual Access Method (VAM) routines (see section 1.1.5.3) that allow O/S batch programs to access ORVYL files using the same routines as are used on-line.

The first part of this chapter describes six large task-oriented groups of subroutines: SRCHREC, ADDREC, DELREC, RPLREC, ATCHFILE, and DTCHFILE. The second describes the various building block subroutines that either are used alone or reside in one of the large task-oriented modules. These building block subroutines are called Basic File Services. Both the basic subroutines and the task-oriented modules are documented in greater detail in Appendix M. Appendix H, Dummy Sections, may be referred to for more detailed information on the format of access method work areas in user virtual memory.

6.2 TASK-ORIENTED SUBROUTINE GROUPS

SRCHREC, ADDREC, DELREC. RPLREC, ATCHFILE, and DTCHFILE are modules constructed as hierarchies of basic file services subroutines. Each component subroutine is itself discussed in section 6.3.

6.2.1 SRCHREC

This module is called to retrieve a particular record of a given type. The record whose key data element value must be found is located regardless of whether the record is in a tree- or slot-structured data set, or whether the record has been removed to the residual data set or not. Figure 36 shows the structure of the SRCHREC subroutine hierarchy.

6.2.2 ADDREC

This module is called to add a new record of a given type. ADDREC locates the key data element value in the new record and creates the node or slot in the data set for the given record type. If removal or splitting to the residual data set is necessary, ADDREC will perform these functions. Figure 37 shows the structure of the ADDREC subroutine hierarchy.

6.2.3 DELREC

This module is called to delete a record of a given type from a user file. The module logically deletes the node or slot in the record type data set, and physically deletes all removed portions and split segments from the user residual data set. Figure 38 shows the structure of the DELREC subroutine hierarchy.

6.2.4 RPLREC

This module is called to replace a record of a given type with a new record. RPLREC locates the existing record with the given key data element value and replaces the index node and any residual entries that may exist. Figure 39 shows the structure of the RPLREC subroutine hierarchy.

6.2.5 ATCHFILE

This module is called during the SELECT <profile name> process in order to attach the data sets of the user file that has been given that profile name and account number. The master data set is attached and accessed. The contents of the master data set are used to initialize a portion of user storage and to set up the characteristics tables in core. ACSENTRY and DSATTACH (see below) are among the subroutines called to assist in performing these functions.

6.2.6 DTCHFILE

This module is called to detach a user file, either because a new SELECT command was issued or because the LOGOFF command was issued. Data set updating and user table cleanup are performed before each attached data set is detached. DSDETACH (see below) is the subroutine called to perform the actual detaching of file data sets.

6.3 BASIC FILE SERVICES SUBROUTINES

These subroutines are used either as components of higher-level task-oriented modules or alone to perform required file services functions.

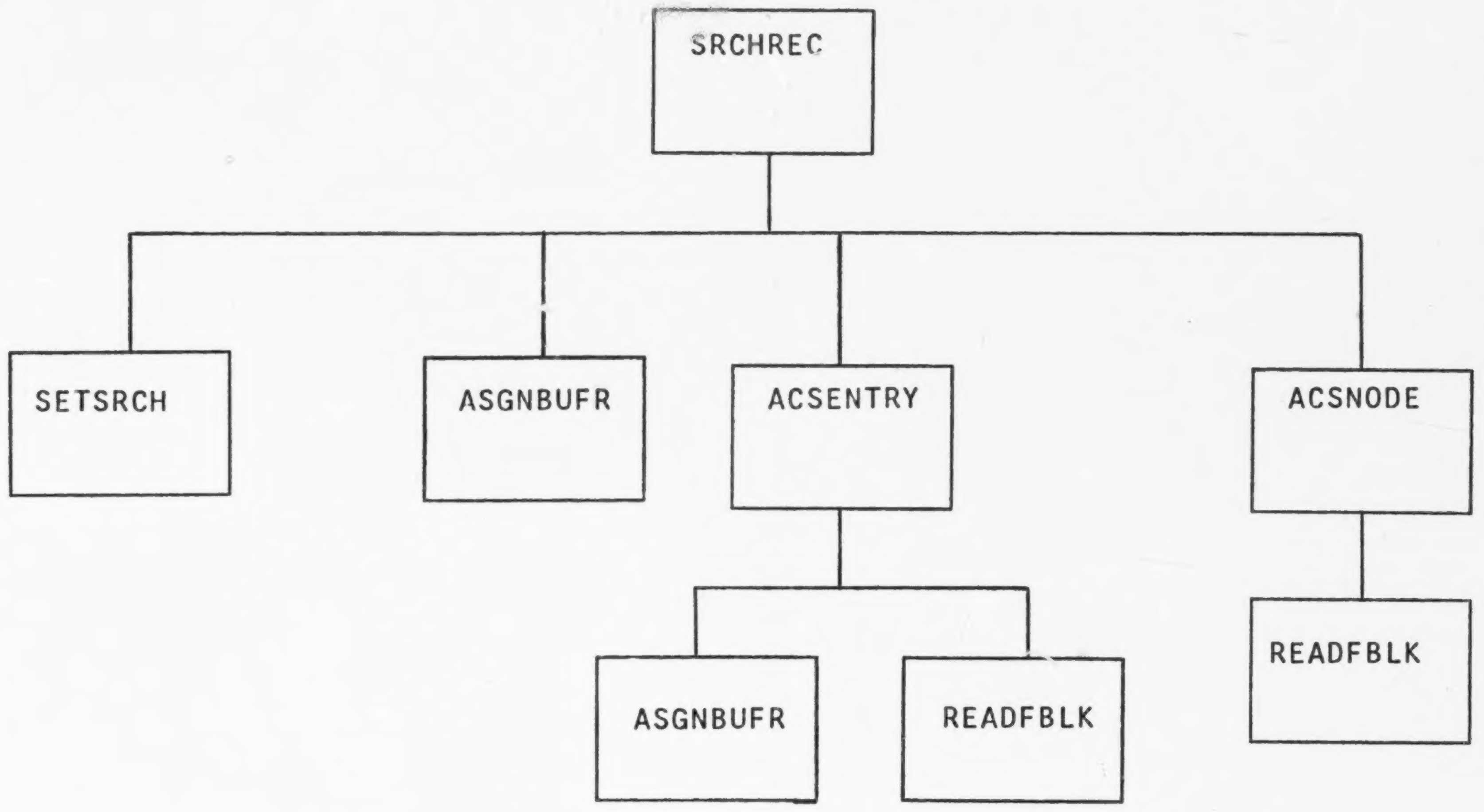

SRCHREC
SETSRCH
ASGNBUFR
ACSENTRY
ACSNODE
ASGNBUFR
READFBLK
READFBLK

Figure 36. SRCHREC Subroutine Hierarchy

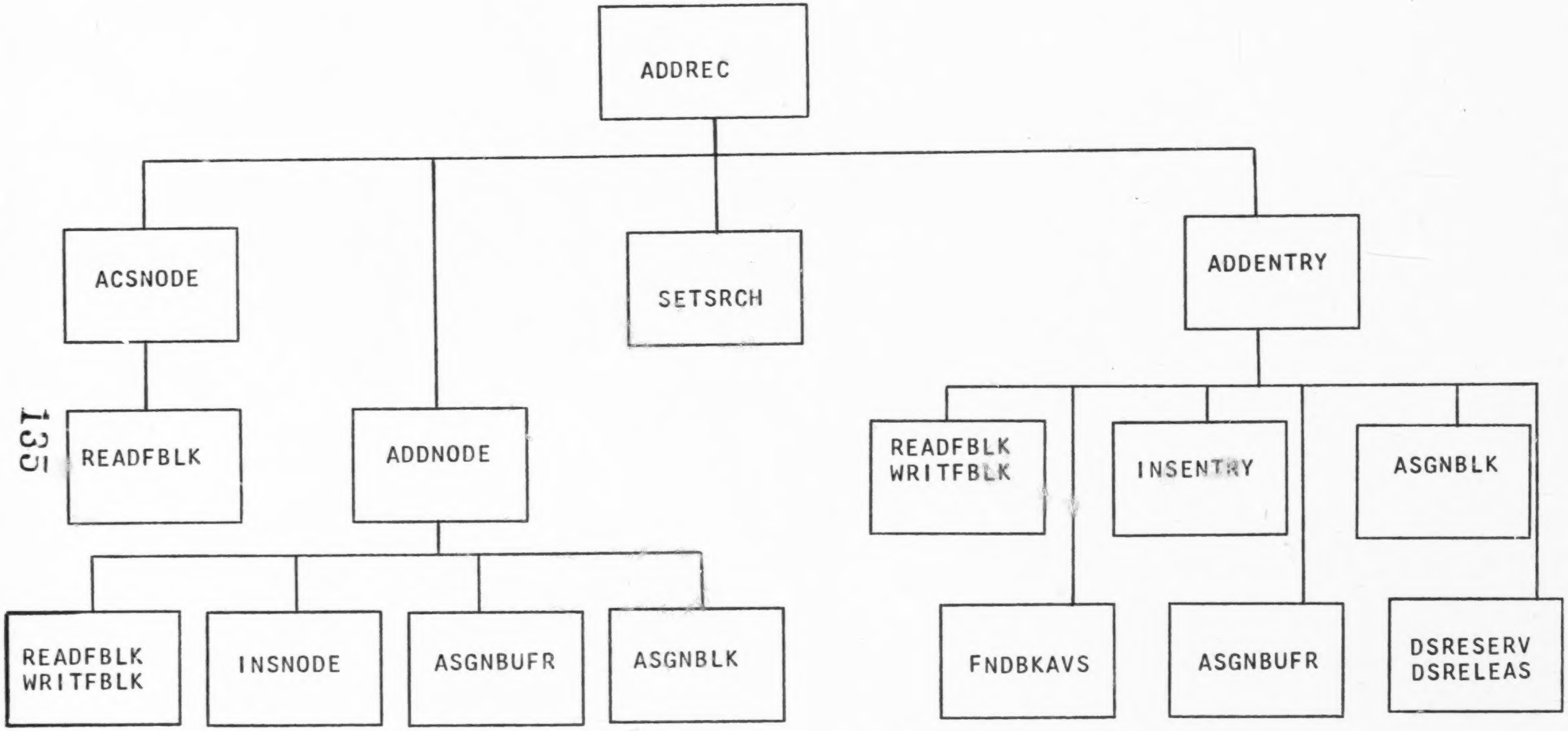

ADDREC
ACSNODE
SETSRCH
ADDENTRY
READFBLK
ADDNODE
READFBLK
WRITFBLK
INSENTRY
ASGNBLK
READFBLK
WRITFBLK
INSNODE
ASGNBUFR
ASGNBLK
FNDBKAVS
ASGNBUFR
DSRESERV
DSRELEAS

Figure 37. ADDREC Subroutine Hierarchy

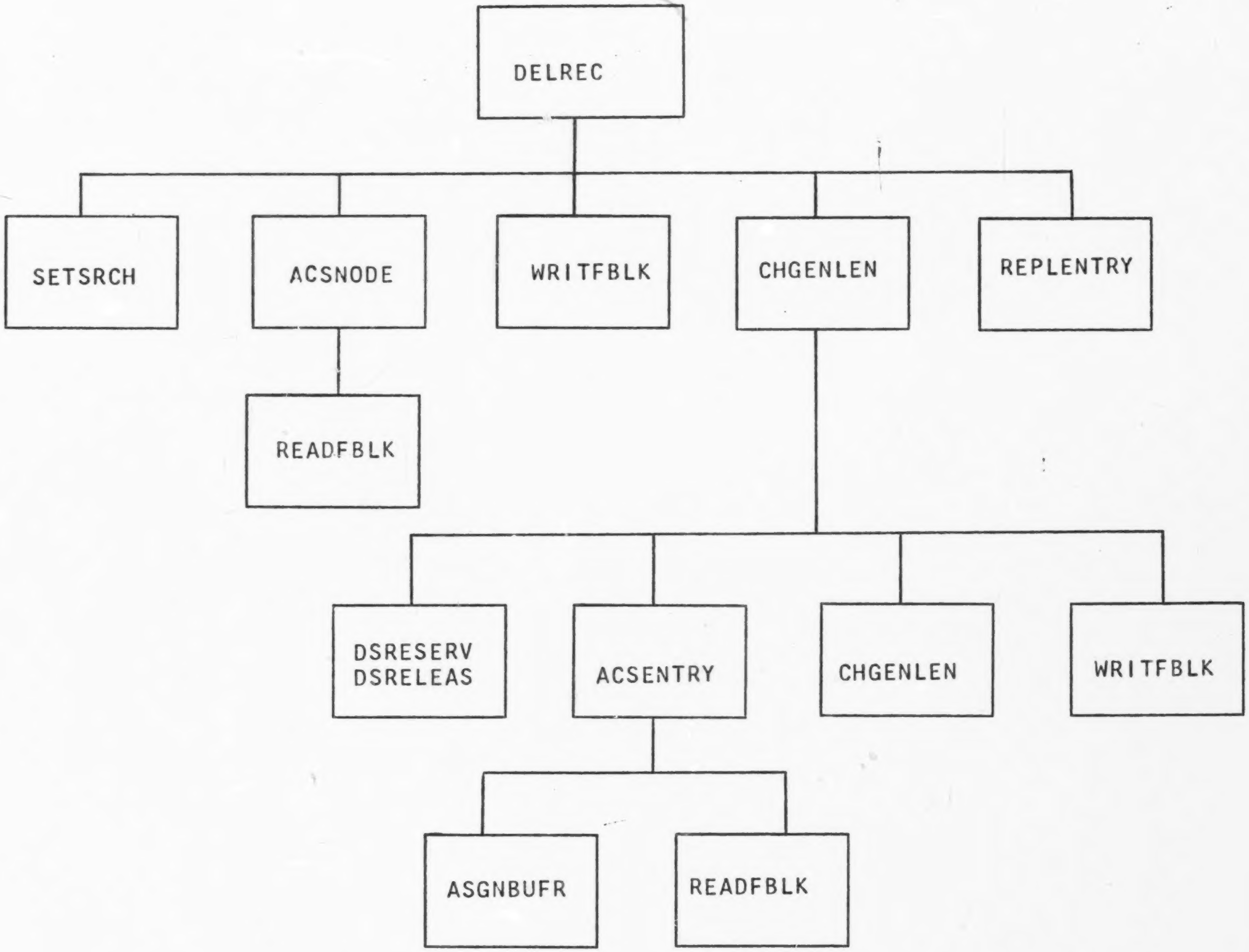

DELREC
SETSRCH
ACSNODE
WRITFBLK
CHGENLEN
REPLENTRY
READFBLK
DSRESERV
DSRELEAS
ACSENTRY
CHGENLEN
WRITFBLK
ASGNBUFR
READFBLK

Figure 38. DELREC Subroutine Hierarchy

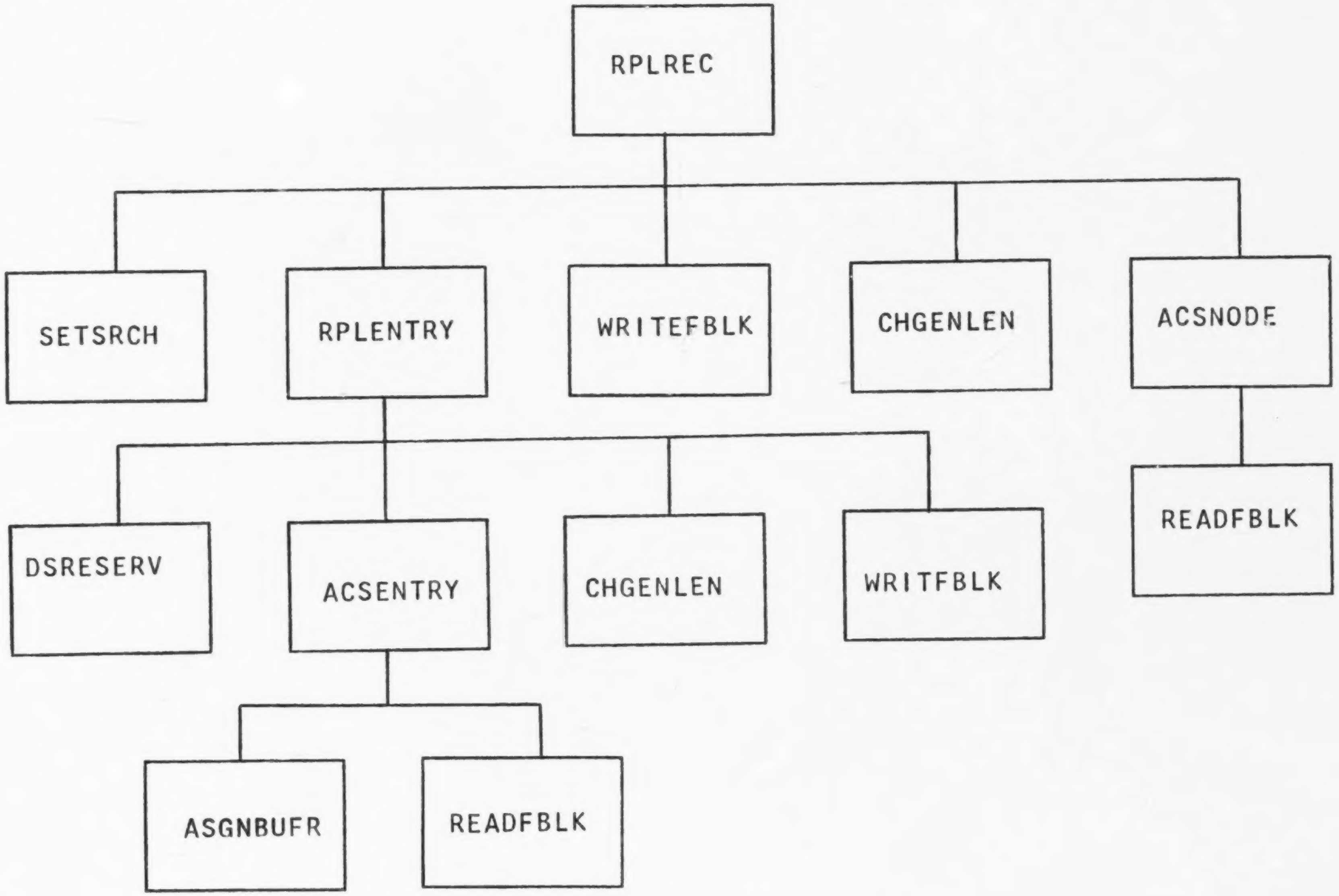

RPLREC
SETSRCH
RPLENTRY
WRITEFBLK
CHGENLEN
ACSNODE
DSRESERV
ACSENTRY
CHGENLEN
WRITFBLK
READFBLK
ASGNBUFR
READFBLK

Figure 39. Record Replacement Subroutine Hierarchy

6.3.1 Data Set Attaching and Detaching Subroutines

These subroutines are called whenever it is necessary to attach or detach a data set within a user or system file set.

6.3.1.1 DSATTACH. This subroutine is called to attach a data set of a user or system file set. The ORVYL interface routine ATCHF is called (see section 2.1.7), and the contents of the device identifier cell are stored in the DILIST entry (see Appendix N) that corresponds to the data set ID number.

6.3.1.2 DSDETACH. This subroutine is called to detach a data set of a user file. After the proper DILIST entry for the data set has been passed to the ORVYL interface routine DTCHF, the entry in DILIST is set to zero to prevent further usage.

6.3.2 Node Manipulation Subroutines

The subroutines described in this section are used to add nodes to a tree-structured data set and to access nodes using their unique key data element values.

6.3.2.1 ACSNODE. This subroutine is called to scan a tree-structured data set for a match of a given key data element value. The located matching node is returned to the caller. If a match was not found, the place in the tree where a node with the given key data element could be inserted is located. READFBLK (see section 6.3.5.1) is called to put the file block involved into core storage.

6.3.2.2 ADDNODE. This subroutine is called to add a new node to a tree-structured data set. It is assumed that ACSNODE has previously been called to locate the block and trailer positions where the new node is to be inserted. The basic subroutines INSNODE, READFBLK, WRITFBLK, ASGNBLK, and ASGNBUFR are called to assist in the procedure.

6.3.2.3 INSNODE. This subroutine, called by ADDNODE, is used to insert a node into a tree data set block that is residing in a core buffer.

6.3.3 Entry Manipulation Subroutines

Entries are records stored in blocks of a residual or a slot-structured data set. Entry manipulation subroutines are usually used for accessing removed or split data in the residual data set, but they are not limited to this. These subroutines may also be used to access records residing in special data sets, such as the master data set or system data sets.

6.3.3.1 ACSENTRY. This subroutine is called to locate an entry within a given data set of a file. Special options may be used to specify whether additional record segments are to be read into user memory along with the requested entry or whether updating entries from the batch queue (see section 7.3.2) are to be accessed instead. READFBLK and ASGNBUFR are called by this subroutine to assist in assigning buffer space and initiating reads.

6.3.3.2 ADDENTRY. This subroutine is called to add a new entry to a non-tree data set. The entry may be added either to existing available space within a block or to an empty block that must be assigned. If the entry data is too large for one block, overflow entries are created in other blocks until all the data has been stored. INSENTRY, READFBLK, WRITFBLK, DSRESERV, DSRELEAS, ASGNBLK, ASGNBUFR, and FNDBKAVS, are called by this subroutine to accomplish lower-level functions.

6.3.3.3 INSENTRY. This subroutine is called to insert an entry into a non-tree data set block. The block must be residing in user memory.

6.3.3.4 RPLENTRY. This subroutine is called to replace an entry in a non-tree data set. If the new entry differs in length from the original, adjustments will be made. All entry overflow problems associated with replacing an entry are handled by this procedure.

6.3.4 Data Element Access Subroutines

These subroutines serve to locate data elements within a record either by their structure element number and their occurrence numbers (which of several occurrences is being referred to) or by their structure element number and their values.

6.3.4.1 ACSELEM. This subroutine is called to access a data element within a record structure, given the structure element number. Optional data elements may be accessed in various ways. If an option flag is set and the element does not exist, a pointer will be set to the location of the element if it had existed.

6.3.4.2 INITVAL. This subroutine is called to initialize certain values in user core (see Appendix H) for accessing a data element by value. The input parameters are: the address of the current structure, the structure element number in the current structure, and the displacement of the start of element values in the file block.

6.3.4.3 GETVALUE. This subroutine is called to locate a data element value within the current structure. It is assumed that both ACSELEM and INITVAL have been previously called, using the read option flag.

6.3.4.4 PUTVALUE. This subroutine is called to store a particular value of the current data element within the current structure. It is assumed that both ACSELEM and INITVAL have been previously called, using the store option flag.

6.3.4.5 PUTVEND. This subroutine is called at the conclusion of a sequence of PUTVALUE calls for a data element. PUTVEND stores the total value length header and value count fields, and ensures that the optional element bit is set if the data element is optional (see section 5.2).

6.3.5 Input and Output Subroutines

6.3.5.1 READFBLK. This subroutine reads a block from a user file data set. Basic error checking is done on the block header to ensure validity. As input parameters, the subroutine must be passed the data set number (record type--see section 4.3.2), the block number of the block to be read, and a buffer address.

6.3.5.2 WRITFBLK. This subroutine writes a block out to a user file data set. As input parameters, the routine must be passed the data set number (record type), the block number of the block to be written, and the buffer address.

6.3.6 Data Set Lockout

During the critical period after a record has been read in for update and before it is written back out, the data set involved is put under exclusive control. When the sequence is completed, the data set is released from exclusive control. In both routines, DSRESERV and DSRELEAS, the data set number is passed as an input parameter.

6.3.7 Allocation Functions

From time to time, as the system is operating, it is necessary to locate available resources (such as virtual memory, empty space in a disk data set, etc.) and to allocate those resources to the user then controlling the system. The following routines are called to perform such functions.

6.3.7.1 ASGNBLK. This subroutine is called to assign and initialize a new block of already allocated space in a data set. The block is initialized as an empty block for the type of data set given. The cell NXTBLK (see section <fill in later>) is used to assign the block number. NXTBLK is incremented by ones. No input or output takes place with this subroutine; the initialization process occurs in the buffer only.

6.3.7.2 ASGNBUFR. This subroutine is called to assign a buffer area in the user logical memory for use by other file service procedures. The routine in turn calls GETCORE, an ORVYL interface (see section 2.1.4), requesting a 2,048-byte extension to the user area.

6.3.7.3 FNDBKAVS. This subroutine is called to locate a block within a data set with sufficient space to satisfy the user request. In the case of residual data sets, the available space block is read into core and the appropriate available space chains are examined for the smallest amount of space that will satisfy the request. Incorrect chain elements (see section 5.4.1) are moved to the correct chains if they are at the tops of chains when found. In the case of data sets other than residual data sets, the next available block is assigned if the current block cannot accomodate the request.

6.3.7.4 INITFILE. This subroutine is called when a new file is to be created. All the data sets needed for the file are created and initialized. The initial record in the master data set must be created and made available to INITFILE as an input parameter. (This record holds the record characteristics for the file.) DSATTACH, ADDENTRY, WRITFBLK, ASGNBLK, and ASGNBUFR are all called by this subroutine.

6.3.8 Miscellaneous Subroutines

6.3.8.1 SETSRCH. This subroutine is called to set up the addresses for a given record type in the record and data element characteristics table in order to search a tree-structured or slot-structured data set.

6.3.8.2. SETSTRCT. The set structure subroutine is called to initialize the structure processing table entry (see Appendix N) in order to prepare for accessing values within a given structure. A required input parameter is the structure number of the given structure. This parameter is zero if the structure is at the top (i.e. record) level.

6.3.8.3 CLOSTRCT. This subroutine is called at the conclusion of all data element processing within a given structure. Clean-up work is performed on the structure table entries and the structure level is decremented back to the next higher-level structure.

CHAPTER 7

SPIRES SYSTEM SUPPORT FUNCTIONS

7.1 INTRODUCTION

The modules and programs described in this chapter are crucial to the smooth functioning of the SPIRES II system--indeed, without them the on-line system would cease to operate. These routines perform file maintenance (updating), error diagnosis, restarting and recovery, and system administration functions. They fall into three operating categories: on-line functions (master commands), O/S batch functions, and ORVYL user program functions.

7.2 THE THREE OPERATING CATEGORIES

Figure 40 lists the various system support functions by module or program name, and shows the category or categories they fall into.

7.2.1 Master Commands

This categorizes functions that can only be called through a SPIRES master terminal. A master terminal is so designated when it is logged on under the SPIRES system account number (SNNN), which is programmed into the system as a constant. At any given time, two persons are responsible for the master terminal and the system account number. The keyword should be changed often (every day, preferably) to avoid weakening security.

When the SPIRES system comes up, the subprocessor area (see 1.1.5.2) contains binary zeros. The master terminal operator logs on into SPIRES and gives the command "MASTER." The system checks to ensure that the user is logged on under the system account number. If he is, then the master terminal flag is set to X'80,' indicating that the system is not ready to receive other users. If other users attempt to log on while the master terminal flag is set to this state, they are returned to WYLBUR with the message "SPIRES NOT IN SERVICE." The master terminal operator may now issue warmstart commands, disable commands, or any of the other commands (see below) that can only be issued when no other users are logged on the the system. When the master terminal operator is satisfied, he issues a PROCEED command, which changes the status of the master terminal flag to X'FF' and unlocks the system for other users. No master terminal is allowed to log on if another master terminal is already logged on.

Module or Program Name	Master Command	O/S Batch	STSM User Program
BATBUILD		X	
DEFUPDT		X	
VALIDATE	X	X	X
FILE LIST	X	X	X
DSZAP	X		
FULDUMP		X	
FULRES		X	
RECOVER		X	
WARMSTRT	X		
PASSREC		X	
DISABLE	X		
AVSPREC	X		X
FILEDEF			X
IDXREBAL		X	
DISKMAP	X		X
STAT	X		X
MESSAGE	X		
ENABLE	X		
INHIBIT	X		
KILL	X		
MAGIC WORD	X		

Figure 40. Utility Support for SPIRES II

7.2.2 O/S Batch Programs

The programs in this category are run in the O/S batch partition because they must have access to magnetic tape or because they require O/S disk data sets as input. They access ORVYL data sets by using the virtual access method (VAM), which permits SPIRES II access method routines to be used in the batch partition with little or no modification.

7.2.3 ORVYL User Programs

The programs in this category are run under ORVYL as user programs for one of the following reasons:

economy;
availability (the O/S batch partition may
not be available);
convenience of development there.

Of the programs that Figure 40 shows to be available in more than one category, it may be that only one version will be operational on day one of the system. In general, this version will be the ORVYL user program.

7.3 THE FILE MAINTENANCE FUNCTIONS

These programs perform all regularly scheduled file updates. They run exclusively in the O/S batch because of their dependence on magnetic tape. The batch build program has the additional requirement of needing O/S disk data sets.

7.3.1 BATBUILD

The batch build program accepts a sequential stream of additions, deletions, and replacements in user files. The update stream for each file is stored in an O/S input data set in a card image. The data in these data sets conforms to the syntax rules of the SPIRES II external format. A user informs the system of the existence of a BATBUILD data set by issuing the command "BATCH <dsname> <bin number>," where the data set named contains the build input. When this command is recognized, the system assembles a record consisting of the user's account number, name, bin number, data set name, and the profile selected at the time the command was issued. The record is then written into the system file set SNNN.BUPDT, which consists of a master data set and one slot-structured data set.

When BATBUILD executes, it opens the system data set SNNN.BUPDT using VAM (see section 1.1.5.3), and reads a header record. This header gives sufficient information for BATBUILD to select a user profile, to open the O/S input data set, and to process the update using exactly the same update semantics as are used on-line. A hard-copy listing for the entire program run is produced that shows each data element value affected, the action taken on each, and any error diagnostics. When the end of a file is reached, the next record from SNNN.BUPDT is read, and the process is repeated for the next file. The format of the hard-copy listing includes page separators giving user name and bin number, to allow the listings to be distributed to the users the next day. As each file is updated, the julian date is entered in the appropriate day slot in the update table in block two of the file's residual data set.

As the build input for each file is read and processed, the input records are also written out on magnetic tape, with the SNNN.BUPDT record serving as a header for each file group. The tape will have been initialized with the volume serial number "BLDnnn," where nnn is the julian date of the build run. This date is also entered in the update table in block two of the file's residual data set. When recovery is necessary, a full restoration is made, and RECOVER (see section 7.5.4) requires mount messages on all tape volumes in which build input has been processed since the last full dump. The volume serial numbers are reconstructed from the julian dates in block two of the file's residual data set.

7.3.2 DEFUPDT

The input for the deferred update program is the contents of the deferred update queue data set "SNNN.DU01." This data set contains the day's update transactions that were entered on-line, translated into internal format. There are two identical copies of the deferred update queue, SNNN.DU01 and SNNN.DU02, kept on different channels and devices. Should an input or output error occur while SNNN.DU01 is being read, DEFUPDT will switch to SNNN.DU02 with no intervention. The probability of both data sets failing simultaneously for the same record is very slight.

The provisions in DEFUPDT for recovery are almost identical to those built into BATBUILD. The input is passed off to a magnetic tape with a specially generated volume serial number, "DEFnnn," where nnn is the julian date. This number is also entered in the day's slot in the update table in block two of the file's residual data set. The program RECOVER can reconstruct the file in the same way as it would using a BATBUILD tape.

7.4 ERROR DIAGNOSTIC ROUTINES

These routines will be used in locating file problems caused by software errors and hardware failures.

7.4.1 VALIDATE

There are two versions of this program. VALIDATE I will eventually exist in all of the three categories. It checks the validity of all the file blocks of a given file, and all the entries and nodes within those blocks. It does not check the validity of record pointers that point across data set boundaries. VALIDATE II performs the same checks that VALIDATE I does. In addition, it verifies all pointers. Since such an operation will doubtless involve sorting, an operation that cannot be performed under ORVYL, VALIDATE II will run under O/S in batch mode. From both versions of the program a listing will be produced of all anomalies found in the specified file. Blocks and nodes or entries found to be in error will be listed out in hexadecimal format where appropriate.

7.4.2 FILELIST

This program will list files or portions of files either logically or physically. If a logical listing is requested, then all the records of a specified record type within a specified key data element value range will be listed in external format, along with all the access records that refer to them. Storage information accompanies each record. If a physical listing is requested, then all the block numbers within a stated number range in the specified data set will be listed in hexadecimal format.

7.5 RESTART AND RECOVERY

The routines that make up RESTART and RECOVER vary from simple on-line master commands to special versions of large O/S batch programs, which have been altered to perform recovery.

7.5.1 DSZAP

This module will eventually exist in all of the three categories. It will accept as input parameters the file name, a data set number (0 for residual, -1 for master, 1, 2, 3...n for record type), a block number, an offset from the beginning of the block, an optional hexadecimal string to be verified, and a hexadecimal replacement string. The program will execute a DSRESERV (see section 6.3.6), read the specified block, compare the verification string (if there is one) to the string at the given offset, and, if the two are equal, replace the string at that offset with the replacement string.

7.5.2 FULDUMP

The FULDUMP program operates either on single files, on a specified subset of SPIRES files defined in SNNN.FILES, or on all SPIRES files listed in SNNN.FILES. FULDUMP copies a specified file, data set by data set, block by block, onto a magnetic tape whose length (1,200 or 2,400 feet if a single reel is used, 2,400 feet if more than one reel is used) is selected according to the total number of blocks in the file's data sets. Before block two of the file's residual data set is dumped, the julian date will be written into the "dump" slot of the update table. The julian date will also be used to create a data set name in the data set header on the magnetic tape: "DMPnnn." The volume serial number of the tape volume containing the dump will be the file name of the file being dumped.

7.5.3 FULRES

This program operates on a single specified file. The input, of course, is the magnetic tape produced in a previous run of FULDUMP. The input parameters are the julian date to be used (the date of the file dump), the file name, and whether the file still exists on disk and is to be overwritten, or whether it is to be completely reallocated. If the file is to be reallocated, a growth space percentage must be provided as a parameter for FULRES. If no julian date is provided and overwriting is desired, a call is made for the last dump version using the julian date stored in block two of the file's residual data set. If no julian date is provided and there is no version available on disk, the operator will be prompted for a julian date.

7.5.4 RECOVER

This program is a special combination of BATBUILD and DEFUPDT, designed to be run after FULRES has restored a file to some previous correct version. RECOVER looks into a file's update table to determine the days of update activity in the period since the file was dumped. If on a particular day either build or deferred update activity occurred, mounts messages are issued for the build or deferred update tape for that day. The update information that applies to the file undergoing recovery is located and copied. This process is repeated until the file is current again.

7.5.5 WARMSTRT

This routine is called by master command only; and it must be called just after the system comes up after a crash, before any users are allowed on the system. WARMSTRT accesses the data set SNNN.FILES. For each SPIRES file listed therein, the residual data set block 0 (the supplemental write block--see section 5.4.2) is read and the first and last words there are compared. If they are equal, the corresponding data set record will be read, and its first and last words compared. If these are unequal, then the supplemental write block is used to overwrite the data set block. If the data set record first and last words are equal, then the supplemental write block is compared with the data set block. If these two are unequal, the supplemental write block is used to overwrite the data set block. Each time a data set block is thus overwritten, a message is written to the master terminal.

7.5.6 PASSREC

This program can be used to regenerate access records consisting only of passed information. Output in the form of nodes or slots is written on magnetic tape, which will then be sorted into key data element value sequence and input to the TREREBAL program (see section 7.6.1).

7.5.7 DISABLE and ENABLE

These routines are called by master command only. The file name specified in the DISABLE command is placed in the subprocessor communications area. Since all attempts to attach a file are checked against this area, users are kept from gaining access to a downed file during a recovery period. The ENABLE command removes the specified file name from the disabled file table.

7.5.8 AVSPREC

This program regenerates block one (the available space table) of a residual data set in a specified file or series of files. Block one is read into user memory, and blocks 3-n are read sequentially. As each block is read, the available space trailer is located, and the amount of available space is used to determine which available space chain is appropriate for the block. After a chain is chosen, the pointer in the available space table is placed in the FBHDAB field of the block header (see Figure 27). The block number of the block--FBHDBK--is moved to the appropriate slot in the available space table. Counts are maintained on the number of blocks in each chain. When the last block in the residual data set is read and chained, a listing of counts is produced, and block two is rewritten to the residual data set.

7.5.9 MESSAGE

This master terminal command causes the two-digit message code specified in the master terminal command to be placed in the subprocessor communications area. This area is checked periodically, and if a new message code is found there, the data set SMNN.MESSAGE is accessed, and the appropriate predefined message is put out at the user terminal. Codes 0-254 are for predefined messages. Code 255 is for a nonpredefined message placed by the master terminal operator in slot 255 of the SMNN. MESSAGE data set. Such a message would be used when it is necessary to broadcast a message not covered by the predefined set.

7.5.10 INHIBIT

This master terminal command prevents any new users from logging onto the system by setting the LOGON flag to X'00.'

The flag is checked by the subprocessor each time a new user attempts to log on.

7.5.11 KILL

This master terminal command places an X'80' in the LOGON flag of the communications area. The subprocessor periodically checks this flag; if it contains an X'80', the user is summarily returned to WYLBUR, with profound apologies put out at his terminal.

7.5.12 MAGIC WORD

This master terminal command places an X'A0' in the LOGON flag. The parameter that accompanies the command verb is moved to the "magic word" field of the communications area. All users attempting to log on when the flag contains an X'A0' are prompted for a magic word, which must match the one in the communications area. If it does not, the user is returned to WYLBUR.

7.6 AIDS TO SYSTEM ADMINISTRATION

The following programs and routines are necessary for the day-to-day administration of the system. They include a tree data set rebalancing program, a disk mapper, and a statistics report generation program.

7.6.1 TREREBAL

This program reads node input from one of two sources: a sorted magnetic tape from PASSREC (see 7.5.6), or one from the tree data set itself. The tree is rebalanced in the way described in section 4.4.3.

7.6.2 DISKMAP

This program produces two types of output. The first, given a disk volume ID or a series of ID's, is a map of the contents of the volume(s) by ascending disk address. The second consists of a listing by specified file name of the physical locations of all data sets and their extents.

7.6.3 STAT

This program will cause a listing to be produced of the day's accumulation of statistics. These statistics will contain information about the frequency of file accesses, module usage, command usage, etc. Another listing will be a tabulation of all error diagnostics issued by the system. At the end of the week, month, and quarter, summary reports will also be produced.

APPENDIX D

Preprints/Anti-Preprints: SLAC Library Monitors Underground Physics Press

Louise Addis

(Reprinted from The SLAC News, 20, June 2, 1971, 2-3.)

Preprints are the underground press of the particle- physics world. For the past three years, the SLAC Library's weekly newsletter "Preprints in Particles and Fields (PPF)" has been providing that world with a popular and reliable master key to its preprint press rooms.

But what are these slightly clandestine preprints? What indeed are anti-preprints? Why is a PPF needed to keep track of them all?

Preprints look innocent enough, a modest sheaf of mimeographed, dittoed, or multilithed sheets locked together by a staple or two. Despite titillating undergound-sounding titles about "Degenerate Daughters," "Ghosts and Gotterdaemmerungen," Two- and Three-Body Problems", reading reveals uniformly benign texts unintelligible to anyone but particle physicists. A few sport fancy covers (like the Lemonade and Orangeade series from Cal Tech) but most are as plain as the hundreds of other documents in the bulging boxes and bags of mail delivered daily to the SLAC Library. But to Rita Taylor, SLAC Preprint Librarian, 50 to 100 items in each week's heap are special. Using, one suspects, ESP or other exotic devices, Rita quickly sorts them from the piles of other material, checks to see that they are not repeats, then launches them on the way to announcement in the next PPF.

These are the preprints. They report the latest experimental and theoretical brain flashes in particle physics, and are being precirculated by their authors at the same time that a manuscript is being submitted for more formal but dilatory publication in a journal. If after months or even years, such a paper finally achieves immortality in the pages of a journal or book, it will be transformed into an "ANTI-PREPRINT", proclaimed in a special green section of PPF, and the original preprint discarded. (A reprint or offprint, by the way, is the exact opposite of a preprint, since it is a copy of an article after it has been published.)

Like many other products of underground presses, preprints are not for sale, but are obtained by being on mailing lists, by knowing somebody else who is, by having a library that makes a real effort to collect them, or by finding out about a particular item in time to write the author for a copy.

Until recently most libraries scorned preprints and most preprint authors made up mailing lists that included people they knew and famous physicists in large laboratories. Less known physicists in out of the way places complained bitterly they couldn't keep up because they didn't get preprints and couldn't even find out about them. Well known physicists complained that their mail boxes were jammed with worthless papers they didn't have time to read. Journal editors worried about the threat of the preprint free-press to the integrity and circulation of their journals, and wasted time and temper trying to run down published versions of preprint references. The work of chronicling the weekly influx of preprints was expensively and imperfectly duplicated by preprint secretaries in countless physics departments. Everyone cried out against the burgeoning circulation of "junk." Preprints, though obviously a vital communication link among physicists, seemed by their very nature defective in the role.

This perplexing preprint paradox (the preprint perplex) was discussed a lot during the 1960s and some elaborate proposals made. Several preprints dealt at length with the question of how to deal with preprints, and a somewhat acrimonious debate developed about the merits of trying to centralized preprint distribution, a proposal which many thought would lead straight to preprints of preprints, to proliferation rather than containment of "junk." Nothing, however, was actually done.

Finally, in 1968, SLAC's Director W.K.H. Panofsky and LRL's Art Rosenfeld were elected Chairman and Secretary of the new Division of Particles and Fields (DPF) of the American Physical Society. Under their leadership DPF formed an alliance with three SLAC Librarians, Louise Addis, Bob Gex, and Rita Taylor, to do something about preprint communication for the whole particle physics community.

The SLAC Library, since its foundation in 1962 on a stack of dusty preprints and some coaching from a CERN Librarian, had been aggressively collecting new preprints and publishing a popular authoritative weekly list of them for SLAC physicists. As years passed, more and more SLAC alumni requested that the SLAC Preprint List be mailed to them at their new institutions. It seemed evident that the simplest, cheapest, and most practical palliative for the preprint perplex was to publish such a list in condensed format, rush it by air to anyone who wanted it, and let him/her or a library acquire the few preprints which were of real interest. Experience had shown that even at SLAC which shelters large numbers of particle physicists, 50-60% of the current crop of preprints are never requested by anyone and that most preprints requested are of interest to just two or three specialists. (At the other extreme, a few important preprints may be of interest to almost everyone.) All preprints must be announced, of course, to enable selection to take place.

The problems of finding preprints later, referencing them after publication, clearing space on desks or library shelves could be solved by including a section called "ANTI-PREPRINTS" listing published preprints with journal, volume, and page references. The SLAC Preprint Librarian had been doing this for years for SLAC and for a few other friendly preprint librarians.

A lightning-fast preprint announcement list coupled with anti-preprint information would complement rather than compete with other physics publications and would not upset the delicate ecology of preprints by slowing them down or overstimulating the distribution of junk. The idea was consistent with a philosophy of preprints as ephemeral documents, rough but speedy.

Master copy for the lists could be produced quickly, easily and elegantly by computer. Since the SLAC Library and its preprints were already participants in a large computerized experimental information system, SPIRES (Stanford Physics Information REtrieval System), very little extra programming would be required.

A proposal was written. In those more affluent days, SLAC soon obtained a special seed-money grant from the AEC to finance printing and mailing such a preprint list to the physicists of the DPF for an 18-month trial period. Computerization was undertaken by Prof. E. Parker's SPIRES group with financing from the National Science Foundation. PPF was on its way.

Since the whole point of preprints is speed, everything about PPF was designed to promote it: speed in production, speed in distribution, and speed in use. A printer was found who could handle the job from repro-ready copy to mailbox within 24 hours. All time-consuming refinements such as elaborate subject classifications, indexes, etc. were rejected. Not quick and dirty, but quick, clean, simple and complete is the motto for PPF.

The first issue of PPF hit the mails in January 1969, and in April all subscribers were queried to see whether the experiment was worth continuing.

The response was overwhelming. More than a thousand subscribers positively wanted to continue getting PPF and hundreds took time to write sometimes lengthy comments and suggestions. Though the PPF staff may have tended to dwell unduly on remarks like "Best thing to happen in physics information in 50 years!" "I have already found one reference which was worth the year's subscription" "Most Valuable publication I get" "PPF is a necessity" " I read it religiously!" and "It's a stroke of genius", clearly PPF met a real need and its future was assured. Several laboratories, including the giant Brookhaven National Laboratory had ceased publishing their own preprint lists and were relying on PPF. One physics department reported reproducing 50-60 copies each week for distribution to faculty and graduate students. Several overseas laboratories made arrangements to reproduce PPF for secondary distribution in their own countries. Even journal editors were enthusiastic about the Anti-Preprints list on practical as well as philosophical grounds. They were using it to exterminate references to old preprints in papers submitted to them for publication.

In July 1970 when the seed-money for printing and mailing ran out, PPF easily became self-supporting. Curently a year's subscription to PPF costs $10/year in the U.S., Canada, and Mexico, $18.50 overseas. It has 531 domestic and 104 overseas subscribers (not counting SLAC), is not copyrighted and is extensively reproduced at its various destinations for further distribution. PPF lists an average of 67 new preprints each week and 100 Anti-Preprints every other week. Each quarter there's a special feature called "PPF Conference Previews and Reviews" which announces future particle physics conferences and explains how to get the proceedings of past conferences. As space permits, various physics events are publicized. DPF notices appear as needed. Subscribers also receive the "Preprint Source Address List" which makes it easier to write to authors for preprint copies. (The third edition dated December 1970 listed 510 addresses), and they may request copies of quarterly cumulations of the Anti-Preprint lists.

All preprints received up to late Wednesday afternoon (and sometimes early Thursday morning) are announced on the week's PPF. (Recently an author phoned to wonder when PPF would get around to listing his preprint which he had sent three weeks ago. It turned out to have been listed the same week he sent it, but it hadn't occurred to him that such was possible.) Coded bibliographic information (authors, titles, report numbers, language, date, number of pages, source) is typed directly into a computer from a time-sharing terminal. The computer sorts, reformats the information and produces the upper-lower case repro-ready copy at the same terminal. On Thursday, the copy is proofed, corrections typed into the computer and final master copy listed for paste-up. The printer picks up the result at 2:00 p.m. PPF goes into the next day's mail and arrives early Monday or Tuesday on physicists' desks all over the world.

The Anti-Preprint list is produced in a similar way after the tables of contents of all new physics journals have been compared with the current preprint collection and published articles matched with corresponding preprints. This is a tricky process since titles change from preprint to article, and sometimes a shortened version of a paper is published in a fast acting "letters" journal while the longer preprinted version awaits full publication in the slower "Physical Review." It's important not to throw away the extensive data in the longer version until it is truly published.

PPF has been managed, edited, and cherished for the three years of its existence by the same team of SLAC Librarians, Louise Addis, Bob Gex, and Rita Taylor, with the indispensible help of Ruth Consolo, lay-out, Rita Glover, computer input, and Barbara Rupp, descriptive cataloging. Bennie Hicks compiles the quarterly section "PPF Conference Previews and Reviews" from her own more complete SLAC publication "Conference Previews."

At SLAC, physicists not only receive a weekly copy of PPF automatically (until recently a special expanded version was published for home consumption), but they have immediate access to all the preprints listed on it. The weeks' preprints are displayed in the Library reading room Monday through Friday where readers may sign up for them or make their own xerox copies. Cards are also filed in the SLAC Library catalog for all the authors of each preprint, so if the recollected author is the twenty-third and his name begins with Z, the preprint can still be located. Subject searching of the preprint collection is available through an on-line information retireval system, SPIRES. A future SLAC NEWS article will describe SPIRES and how it may be used for personally tailored literature searches of the underground press of particle physics.